UiO : Faculty of Mathematics and Natural Sciences
University of Oslo

Numerical Methods for Solving the Fastest Mixing Markov Chain Problem

Yang Kjeldsen Yang

Master's Thesis, Spring 2015

This thesis represents 60 credits and is written for a Master's degree at the Department of Mathematics, University of Oslo.

Acknowledgements

I would like to take this oppertunity to express my gratitude to my supervisor, Geir Dahl, for his guidance and the interestin discussions we had about the thesis. I want to show my appreciation to Torkel Haufmann for taking the time whenever I needed help with mathematical problems. I will thank Tine Venås for her amazing job with finding typos and checking the grammar, my brother Jing for reviewing the content in this thesis, and Fredrik, Pia and Astri for the wonderful company on the 6th floor at Abel. To my parents, for all the love and support.

Yang K. Yang,
Oslo, May 2015

Abstract

Consider a random walk on an undirected, connected graph. On each edge we can set a transition probability to connect two adjacent vertices. The mixing rate of the associated Markov chain to the uniform equilibrium distribution is determined by the second largest eigenvalue in modulus (SLEM) of the transition probability matrix. This problem is called the fastest mixing Markov chain problem (FMMC).

This thesis will cover numerical methods for solving the FMMC problem. We will compare different methods for solving the problem, including the subgradient method and primal-dual interior-point methods. We will provide a complexity analysis of implentation of the methods, and make a comparison with the convex optimization solver CVXOPT written by Andersen, Dahl, and Vandenberghe. Finally, we will also look at small applications of the problem in shuffling.

Contents

CHAPTER 1

Introduction

In 2004, Boyd et al., Boyd, Diaconis, and Xiao published two papers about the fastest mixing Markov chain on a graph and on a path, [2, 3]. The area they studied shows great importance in fields like statistics, physics, chemestry, biology and computer science. For instance, when we play a card game there is many ways to shuffle a deck of cards. Most of the time we want the deck to be shuffled properly in order to make the game as fair as possible. If we want to minimize the time for a shuffle, we can set up the problem as a random walk on a graph, more precisely, the fastest mixing markov chain problem.

In chapter 2, we will give the background theory of the problem we are going to work with in this thesis. First, we will give a brief introduction to graph theory, discrete-time Markov chain and convexity. And then we will continue with forming a semidefinite program and present two types of methods called interior-point methods and subgradient method to solve the problem.

Chapter 3 introduces the fastest mixing Markov chain (FMMC) problem, which is the main topic in this thesis. We will discuss the FMMC problem, why it is a convex optimization problem, by going in to detail of some of the steps. We will also show that it can be formaulated as a specific type of convex optimization problem - a semidefinite program. Finally, we will look at two applications of the problem for motivation and to give insight in the reason we do this.

Chapter 4 is the numerical chapter where we report where we solves the fastest mixing markov chain problem. We will consider different types of graphs, including path, cycle and star graphs. A section will deal with the choice of the step length of the subgradient method. Lastly, we will look at randomly generated graphs for the FMMC problem.

In chapter 5, we will compare the primal-dual interior-point methods with the CVXOPT package for solving the FMMC problem as a semidefinite program. We will look into the implementation of the primal-dual interior-point methods and give an analysis of the complexity.

In chapter 6, we will discuss some future research for the FMMC problem.

The appendix contains the implementations of the subgradient method,

primal-dual interior-point methods, a short introduction to graph representation in NetworkX, and an implementation of solving semidefinite programs using CVXOPT is also included.

A summary of my contribution. A big part of this thesis has been implementation of algorithms. I have implemented all the programs which is contained in the appendices, although, the algorithms are found in different articles, it has been a lot of work testing and running the programs. I have given two algorithms for modeling the FMMC problem as a SDP, solvable for primal-dual interior-point methods and the CVXOPT. The algorithm of a projected subgradient method is stated in [2], which I also have implemented. Most of the test runs, tables and plots are my own work, reference will be stated where I got inspiration.

CHAPTER 2

Background theory

In this chapter we will give a brief introduction to the background theory of the problems we are going to work with. The first section will cover the basics of graph theory where we use the definitions from the book, *Graph Theory*, by Bondy and Murty [4]. After this part, we will introduce Markov chains using the book *Introduction to Probability* by Grinstead and Snell [5], as reference, and continue with a short reminder of convexity and a semidefinite program from Boyd and Vandenberghe [6]. The final part in this chapter covers the interior-point methods and the subgradient method using Boyd and Vandenberghe, Boyd, Xiao, and Mutapcic [6, 7].

1. Graph

We can think of a *graph* as an unordered pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consisting, of a set $\mathcal{V}$ of *vertices* and a set $\mathcal{E}$, disjoint from $\mathcal{V}$, of *edges*, together with an *incidence function* $\psi_{\mathcal{G}}$ that associates with an edge of $\mathcal{G}$ an unordered pair of vertices of $\mathcal{G}$. If edge $e \in \mathcal{E}$ and u and v are vertices such that $\psi_{\mathcal{G}}(e) = \{u, v\}$, then e is said to *join* u and v, and the vertices u and v are called the *ends* of e. The *order* of $\mathcal{G}$ is the number of vertices in $\mathcal{G}$ and the *size* is the number of edges in $\mathcal{G}$. The ends of an edge are *incident* with the edge, and vice versa. Two vertices which are incident with a common edge are *adjacent*, as are two edges which are incident with a common vertex, and two distinct adjacent vertices are *neighbors*.

A graph $\mathcal{G}$ is *connected* if there exists a *path* between any pair of vertices of the vertex set $\mathcal{V}$. An *undirected graph*, is a graph where the direction of the edge is irrelevant, meaning, that if edge $(i, j) \in \mathcal{E}$ then $(j, i) \in \mathcal{E}$ and vice versa, if edge $(i, j) \notin \mathcal{E}$ then $(j, i) \notin \mathcal{E}$. And in this paper we will concentrate on graphs which are connected and undirected.

For the purpose of applying mathematical methods to study their properties, or storing graphs in computers, we consider two matrices associated with a graph, its incidence matrix and its adjacency matrix. Let $\mathcal{G}$ be a graph, with vertex set $\mathcal{V}$ and edge set $\mathcal{E}$. The *incidence matrix* of $\mathcal{G}$ is the $n \times m$ matrix $\mathbf{M}_{\mathcal{G}} := (m_{ve})$, where m_{ve} is the number of times $(0, 1, \text{or } 2)$ hat vertex v and edge e are incident. The *adjacent matrix* of $\mathcal{G}$ is the $n \times n$ matrix $\mathbf{A}_{\mathcal{G}} := (a_{uv})$, where a_{uv} is the number of edges joining vertices u and b, each loop counting as two edges. A more compact way of representing graphs, is to list the neighbors of each vertex v in some order. A list $(N(v) : v \in \mathcal{V})$ of these lists is called the adjacency list of the graph.

2. Discrete-time Markov chain

In this section we will introduce discrete-time Markov chain using Levin, Peres, and Wilmer, Grinstead and Snell [8, 5]. We will look at properties of the Markov chain which will be useful to state the convergence of such chains using Levin, Peres, and Wilmer [8].

Using Grinstead and Snell [5]. We specify a Markov chain by first considering a set of *states*, $\Omega = \{s_1, \ldots, s_r\}$, where Ω is the *state space* of the chain. The idea of the Markov chain is that we start in one of these states and move successively from one state to another. Each move in this process is called a *step*. In a current state s_i, we can associate a probability p_{ij} for the chain to move to state s_j in the next step. The probability p_{ij} is independent, which means that it will not depend on previous states. We call the probabilities p_{ij}, *transition probabilities*. The probability of remaining in the current state s_i for the next step is with p_{ii}. We can represent the transition probability matrix $P \in \mathbb{R}^{r\times r}$ where the p_{ij} denotes the entries of the matrix.

THEOREM 2.1. *Let P be a transition matrix of a Markov chain. The ij-th entry $p_{ij}^{(n)}$ of the matrix P^n gives the probability of the Markov chain, starting in state s_i, will be in state s_j after n steps.*

THEOREM 2.2. *Let P be a transition matrix of a Markov chain, and let π be the probability vector which represents the starting distribution. Then the probability, that the chain is in state s_i after n steps is the i-th entry on the vector*

$$\pi^{(n)} = \pi P^n$$

Both theorems are results found in Grinstead and Snell [5].

We will now propose two properties about Markov chains, which will be necessary to show convergence of the chains to an stationary state. But first, we will define the total variation distance in order to measure two distributions on the Markov chain.

The *total variation distance* is a norm between two probability distributions μ and ν on Ω defined by

$$\|\mu - \nu\|_{tv} = \max_{A\subseteq\Omega} |\mu(A) - \nu(A)|. \tag{1}$$

It is the maximum difference between μ and ν assigned to any subset A of Ω.

A Markov chain is called *irreducible* if for any two states $s_i, s_j \in \Omega$ there exist an integer t such that the $P_{ij}^t > 0$ for the transition probability matrix P. This means that it is possible to get from any state to any other state using only transition of positive probability.

Let $\mathcal{T}(s_i) := \{t \geq 1 : P_{ii}^t > 0\}$ be the the set of times when it is possible for the chain to return to starting state s_i for $1 \leq i \leq r$. A *period* of a state s_i is given by the greatest common divisor of $\mathcal{T}(s_i)$. A chain is *aperiodic* if all the states have period 1.

THEOREM 2.3 (Convergence theorem). *Suppose that P is irreducible and aperiodic, with stationary distribution π. Then there exist constants $\alpha \in (0,1)$ and $C > 0$ such that*

$$\max_{1 \le i \le r} \|P_{i*}^t - \pi\|_{tv} \le C\alpha^t \tag{2}$$

where $\|\cdot\|_{tv}$ is the total variational distance and P_{i*}^t denotes the i-th row of P^t.

The proof of theorem 2.3 can be found in Levin, Peres, and Wilmer [8].

3. Convexity

Here we are going to give a little reminder of the basics of convexity found in Boyd and Vandenberghe [6].

A set A is called *convex*, or a *convex set*, if for any two points, x and y in A, then $\mu x + (1-\mu)y \in A$ where $0 \le \mu \le 1$. To interpret this a set is convex if we can take any two points of the set, draw a line between them, and the whole line is contained in the set.

Next, assume that set A is convex. Then $z \in A$ is called an *extreme point* of a convex set if $z = \alpha x + (1-\alpha)y \in A$ implies that $x = y \in A$ when $0 \le \alpha \le 1$.

Let $f : \mathbb{R}^n \to \mathbb{R}$ and let A_f denote the domain of f. Assume that A_f is convex then we say that the function f is *convex* if $x, y \in A_f$ then

$$f(\mu x + (1-\mu)y) \le \mu f(x) + (1-\mu)f(y)$$

for all $\mu \in [0,1]$.

4. Semidefinite programming

Semidefinite programming (shorten as SDP) is a class of convex optimization problems. We can set up a semidefinite program in this way

$$\begin{aligned} &\text{minimize} && c^T x \\ &\text{subject to} && F_0 + \sum_{i=1}^{m} x_i F_i \succeq 0, \\ & && Ax = b \end{aligned} \tag{3}$$

where $c \in \mathbb{R}^m$ and $F_0, \ldots, F_n$ are $k \times k$ symmetric matrices and $A \in \mathbb{R}^{p \times n}$ (see Boyd and Vandenberghe [6]). A semidefinite program on this form has linear equality constraints by $Ax = b$ and an affine function $F_0 + \sum_{i=1}^m x_i F_i$ which is positive semidefinite.

For the next two sections, we will introduce methods for solving convex optimization problems using Boyd and Vandenberghe [6], Alizadeh, Haeberly, and Overton [9] and Boyd, Xiao, and Mutapcic [7].

5. Interior-point methods

In this section we will cover the basic ideas of the interior-point methods for solving convex optimization problems. We will look at Newton's method which is useful later, in particular, we will look at interior-point methods for solving semidefinite program. The primal-dual interior-point methods we are going to take a closer look at, are the XZ-method and $XZ + ZX$-method, which are two algorithms which can be used to solve semidefinite programs, and can be found in Alizadeh, Haeberly, and Overton [9].

5.1. Newton's method. In this section we will show the Newton's method for an optimization problem in order to get the idea of the application. The method is used for solving convex optimization such as semidefinite programming, and it is applied to find the search direction. We will later in this look at interior-point methods where the Newton's method is applied. First, we will derive the basic idea by looking at an optimization problem and then carry on with the details afterward. To give this short introduction to Newton's method we have used theory from Boyd and Vandenberghe [6].

We start by looking at a convex optimization problem on the form

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{subject to} && Ax = b, \end{aligned} \tag{4}$$

where $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n\times n}$. Assume that the function $f : \mathbb{R}^n \to \mathbb{R}$ and that it is twice differentiable and convex. We also assume that x is a feasible point, that is, it satisfies $Ax = b$. The basic idea of Newton's method is to find a search direction Δx for the purpose of getting closer to the optimal value. To do that, we need to calculate the *Newton step* such that when we update the feasible point x with Δx , $x \to x + \Delta x$.

The way we are going to approach such points is to look at the quadratic Taylor approximation of the function f.

$$T_f(x + \Delta x) = f(x) + \nabla f(x)^T \Delta x + (1/2)\Delta x^T \nabla f(x)^2 \Delta x + R(\Delta x^3)$$

where $R(\Delta x^3)$ is the remainder of the Taylor approximation order 3. For our usage, we will use the approximation where we neglect the remainder and look at the approximation as

$$T_f(x + \Delta x) \approx f(x) + \nabla f(x)^T \Delta x + (1/2)\Delta x^T \nabla f(x)^2 \Delta x$$

By replacing the objective function with the Taylor approximation we can form the new problem as

$$\begin{aligned} &\text{minimize} && F(x^*) \\ &\text{subject to} && Ax^* = b, \end{aligned} \tag{5}$$

where $x^* = x + \Delta x$ and $F(x^*) = f(x) + \nabla f(x)^T \Delta x + (1/2)\Delta x^T \nabla f(x)^2 \Delta x$. Consider the Lagrange function

$$L(x+\Delta x, \lambda) = f(x)+\nabla f(x)^T \Delta x +(1/2)\Delta x^T \nabla f(x)^2 \Delta x +\lambda^T (A(x+\Delta x)-b)$$

In order to find a search direction Δx that maintains the feasilbility of the problem, we will differentiate the Langrange function L with respect to Δx

such that we obtain

$$\nabla_{\Delta x} L(x + \Delta x) = \nabla f(x)^T + \nabla^2 f(x)\Delta x + A^T\lambda$$

For optimality, we set $\nabla_{\Delta x} L(x + \Delta x) = 0$ such that we can solve a linear system with linear equalities to obtain the search direction Δx and the corresponding dual variable λ. The system to solve is on the form

$$\begin{pmatrix} \nabla^2 f(x) & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \lambda \end{pmatrix} = \begin{pmatrix} -\nabla f(x) \\ 0 \end{pmatrix}$$

After we have found the search direction Δx we know that the new point will improve and hopefully get closer to the optimal point. The update is $x \to x + \Delta x$ which is feasible indeed.[6]

5.2. Interior-point methods for semidefinite programming. The article "Primal-dual interior-point methods for semidefinite programming: convergence rates, stability and numerical results" (see [9]) introduces interior-point methods for solving semidefinite programs. In this section we will only take care of two of these methods, which are the XZ-method and the $XZ + ZX$-method. Both algorithms for the methods are described in the article, so here we will focus on the application of Newton's method in the two methods and highlight properties about the iterates from the article.

Let us define some notation before we move on. Define the inner product of two matrices, A and B, as

$$\langle A, B\rangle = \operatorname{Tr} AB = \sum_i \sum_j A_{ij}B_{ij}$$

We look at the semidefinite program on the form

$$\begin{aligned} &\text{minimize} && \langle C, X\rangle \\ &\text{subject to} && \langle A_i, X\rangle = b_i,\ i = 1, \dots, m, \\ & && X \succeq 0 \end{aligned} \tag{6}$$

provided that X is symmetric which we name the primal semidefinite programming problem. The dual of the semidefinite program can be written as

$$\begin{aligned} &\text{maximize} && b^T y \\ &\text{subject to} && \sum_{i=1}^{m} y_i A_i + Z = C, \\ & && Z \succeq 0 \end{aligned} \tag{7}$$

where Z is symmetric.

Now that we have all the notations and definitions we need, we formulate the the optimal conditions of the semidefinite program. To do this, we have to consider primal feasibility, dual feasibility and the complementary slackness conditions. The primal feasibility constraint is $\langle A_i, X\rangle = b_i$ for $i = 1, \dots, m$, the dual feasibility constraint is $\sum_{i=1}^m y_i A_i + Z = C$ and the complementary slackness can be formulated as $XZ = \mu I$ for some $\mu \in \mathbb{R}$, $\mu > 0$, for the central path. For each such μ on the central path we can

associate points (X^μ, y^μ, Z^μ) where X^μ, Z^μ are symmetric matrices of $\mathbb{R}^{n\times n}$ and $y^\mu \in \mathbb{R}^m$.

For semidefinite programs, the points on the central path satisfy the nonlinear equation

$$\begin{bmatrix} \sum_{k=1}^m y_k A_k + Z - C \\ \langle A_1, X\rangle - b_1 \\ \vdots \\ \langle A_1, X\rangle - b_m \\ XZ - \mu I \end{bmatrix} = 0 \tag{8}$$

First we will look at the Newton step which satisfy the dual condition

$$\sum_{k=1}^m \Delta y_k A_k + \Delta Z = C - (\sum_{k=1}^m y_k A_k + Z)$$

and similar for the the primal conditions we have

$$\langle A_i, \Delta X\rangle = -(\langle A_i, X\rangle - b_i)$$

for $i = 1, \ldots, m$. When we derive the Newton step for $XZ = \mu I$ we will do this in two parts. First we will consider the XZ-method and then show the $XZ + ZX$-method.

When we formulated the semidefinite programming in equation 6 we wanted to find a symmetric X, as the X iterates are not symmetric, we would have to update X with the symmetric search direction $(1/2)(\Delta X + \Delta X^T)$ before we continue with the next iteration. Because of additional symmetry step of X, the XZ-method is not a Newton's method, as we consider the Newton step to be derived from solving the nonlinear equation 8. For the Newton step of the XZ-method, it satisfy $X\Delta Z + \Delta X Z = \mu I - XZ$

The $XZ + ZX$-method is a Newton method since we do not require the symmetry step of the X after solving equation 8.

We are now going to show the duality gap between primal and dual of the semidefinite program. The duality gap will give us an indicator of how close we are to the optimal solution. To get optimality, we require that X, y, Z satisfy the following conditions

$$\langle A_k, X\rangle = b_k, \ k = 1, \ldots, m \tag{9}$$

$$C = Z + \sum_{k=1}^m y_k A_k \tag{10}$$

$$X, Z \succeq 0 \tag{11}$$

By calculation we get that

$$\begin{aligned}\langle C, X\rangle - b^T y &= \langle Z + \sum_{k=1}^{m} y_k A_k, X\rangle - \sum_{k=1}^{m} y_k b_k \\ &= \langle Z, X\rangle + \sum_{k=1}^{m} y_k \langle A_k, X\rangle - \sum_{k=1}^{m} y_k b_k \\ &= \langle Z, X\rangle + \sum_{k=1}^{m} y_k (\langle A_k, X\rangle - b_k) = \langle Z, X\rangle\end{aligned}$$

which means that the duality gap is given by

$$\langle C, X\rangle - b^T y = \langle Z, X\rangle. \tag{12}$$

We can use this gap as a stopping criterion for the primal-dual interior-point methods.

6. Subgradient method

In this section we will look at the basics of the subgradient method and highlight important results from *Subgradient Methods* by Boyd, Xiao, and Mutapcic [7]. We will look at the iteration step of the method and show in detail why this method works by deriving the convergence. We will also explain shortly the projected subgradient method, which is a variant of the subgradient method, later in this section.

Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is a convex function. A *subgradient* of a f at x is a vector g that satisfy

$$f(y) \geq f(x) + g^T(y - x) \text{ for all } y \in \mathbb{R}^n \tag{13}$$

From [7], we can formulate the following iteration step for the subgradient method.

$$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}, \tag{14}$$

where the $x^{(k)}$ is the k-th iterative point, α_k is the k-th step size (where $\alpha_k > 0 \; \forall \; k \in \mathbb{N}$) and $g^{(k)}$ is the any subgradient of f at $x^{(k)}$. For a differentiable convex function f, the only choice of the subgradient is $g = \nabla f$ but if we neglect that f is differentiable, we may have more than one choice of choosing the subgradient. The method has some similarities with ordinary gradient methods, but it has also some differences. For example, the step size of the subgradient is determined a priori which according to the gradient method is calculated by applying a line search. The subgradient method is not a descent method, which means that an iteration of the method may not necessarily give a better function value, it may also increase. For the purpose of this, we have to keep track of the best solution for each iteration. A simple approach to obtain this is to let

$$f_{\text{best}}^{(k)} = \min\left(f_{\text{best}}^{(k-1)}, f(x^{(k)})\right) \tag{15}$$

then for the k-th step we have found that

$$f_{\text{best}}^{(k)} = \min_{l=1,\ldots,k} \Big(f(x^1), \ldots, f(x^{(k)})\Big). \tag{16}$$

The challenge with the subgradient method is that it is hard to find good stopping criterion for the algorithm, although $f_{\text{best}}^{(k)}$ is a monotone decreasing sequence we do not know when the optimal value is obtained.

For the step size, we have different types we can use.

(1) Constant step size, $\alpha_k = h$, where h is a constant.
(2) Constant step length, $\alpha_k = h/\|g^{(k)}\|_2$, where $\|\cdot\|_2$ is the Euclidean norm, h is a constant and $g^{(k)}$ is the k-th subgradient.
(3) Square summable but not summable, the step sizes satisfy

$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty \text{ and } \sum_{k=1}^{\infty} \alpha_k = \infty \tag{17}$$

(4) Nonsummable diminishing, the following are satisfied

$$\lim_{k\to\infty} \alpha_k = 0 \text{ and } \sum_{k=1}^{\infty} \alpha_k = \infty, \tag{18}$$

and step sizes that satisfy the conditions above are called diminishing step rule.

We will now take a closer look at the convergence of the subgradient method which can be found in Boyd, Xiao, and Mutapcic [7].

Assume that the optimal value is obtained at a point x^*, which is denoted by $f^* = f(x^*)$ and the iteration step

$$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)} \tag{19}$$

for $k \in \mathbb{N}$. The k-th step subgradient $g^{(k)}$ defined by

$$f(x^{(k)}) - f(x^*) \leq (g^{(k)})^T (x^{(k)} - x^*) \tag{20}$$

By applying the formula for the iteration step in equation 19 and the subgradient step in equation 20, we get that

$$\begin{aligned}
\|x^{(k+1)} - x^*\|_2^2 &= \|x^{(k)} - \alpha_k g^{(k)} - x^*\|_2^2 \\
&= \|x^{(k)} - x^*\|_2^2 - 2\alpha_k((g^{(k)})^T(x^{(k)} - x^*) + \alpha_k^2 \|g^{(k)}\|_2^2 \\
&\leq \|x^{(k)} - x^*\|_2^2 - 2\alpha_k(f(x^{(k)}) - f(x^*)) + \alpha_k^2 \|g^{(k)}\|_2^2
\end{aligned}$$

We use induction on k by repeating the same procedure on $\|x^{(k)} - x^*\|_2^2$ and sum the terms. Recursively, we obtain

$$\|x^{(k+1)} - x^*\|_2^2 \leq \|x^{(1)} - x^*\|_2^2 \ - 2\sum_{l=1}^{k} \alpha_l (f(x^{(l)}) - f(x^*)) + \sum_{l=1}^{k} \alpha_l^2 \|g^{(l)}\|_2^2$$

From the definition of a norm, we have that $\|x^{(k+1)} - x^*\|_2 \geq 0$. Thus, we have that

$$2\sum_{l=1}^{k} \alpha_l (f(x^l) - f(x^*)) \leq \|x^{(1)} - x^*\|_2^2 + \sum_{l=1}^{k} \alpha_l^2 \|g^{(l)}\|_2^2$$

and

$$\begin{aligned}\sum_{l=1}^{k}\alpha_l(f(x^l)-f(x^*)) &\geq \min_{l=1,\dots,k}\{f(x^l)-f(x^*)\}\sum_{l=1}^{k}\alpha_l \\ &= \left[\min_{l=1,\dots,k}\{f(x^l)\}-f(x^*)\right]\sum_{l=1}^{k}\alpha_l \\ &= \left[f_{\text{best}}^{(k)}-f(x^*)\right]\sum_{l=1}^{k}\alpha_l\end{aligned}$$

where $f_{\text{best}}^{(k)} = \min_{l=1,\dots,k}\{f(x^{(l)})\}$ denotes the best solution of k iterations. We combine the last two equations and we will get

$$f_{\text{best}}^{(k)} - f(x^*) \leq \frac{\|x^{(1)}-x^*\|_2^2 + \sum_{l=1}^{k}\alpha_l^2\|g^{(l)}\|_2^2}{2\sum_{l=1}^{k}\alpha_l} \tag{21}$$

and assume that the norm of the subgradient $g^{(l)}$ for $l = 1, \dots, k$ is bounded, that is, $\|g^{(l)}\|_2 \leq M$, where $M \in \mathbb{R}^n$ is a constant. Now we can see that if we choose the step size α_i appropriately we are able to limit the distance between the best solution and the optimal solution. The step size rules that we mentioned earlier in this section can all be used to get convergence. The convergence result and the proofs of this can be found in Boyd, Xiao, and Mutapcic [7].

A variant of the subgradient method is the *projected subgradient method*. The method can be used to solve convex optimization problem on the form

$$\text{minimize} \quad f(x) \quad \text{subject to} \quad x \in A \tag{22}$$

where A is a convex set. In the projected subgradient method define the iteration step as $x^{(k+1)} = \text{proj}_A(x^{(k)} - \alpha_k g^{(k)})$ where proj_A denotes the projection on A.

CHAPTER 3

Fastest mixing Markov chain problem

1. Fastest mixing Markov chain - a convex optimization problem

In the last chapter we established the background theory we need so we can get started with the formulation of the problem using Boyd, Diaconis, and Xiao [2]. We will formulate the fastest mixing Markov chain problem and look at different formulations of the problem in order to solve the problem using convex optimization algorithms.

We consider a connected, undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with vertex set $\mathcal{V} = \{1, \ldots, n\}$ and edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, with $(i, j) \in \mathcal{E}$ and $(j, i) \in \mathcal{E}$. On each vertex, we will also make the assumption that it has a self-loop, i.e, an edge from itself to itself: $(i, i) \in \mathcal{E}$ for $i = 1, \ldots, n$. We can formulate a Markov chain on the graph $\mathcal{G}$ where we let the vertex set $\mathcal{V}$ be the state space of the chain. The state at time t will be denoted with $X(t) \in \mathcal{V}$ for $t = 0, 1, 2 \ldots$. On each edge we associate a transition probability of the graph, where a state moves to a new state, or stays at the same state. The transition probability matrix $P \in \mathbb{R}^{n \times n}$ that describes the Markov chain can be denoted as

$$P_{ij} = \text{Prob}(X(t+1) = j \mid X(t) = i), \ i, j = 1, \ldots, n$$

We call $X(t)$ the state of a Markov chain at time t. The probability of making a transition to state $X(t+1)$ at a time step $t+1$ is only dependent on the previous step $X(t)$ at time t. Let $\pi(t) \in \mathbb{R}^n$ be the probability distribution of the state at time t, such that $\pi(0)$ is the initial probability distribution. The i-th element of the vector is denoted by $\pi_i(t) = \text{Prob}(X(t) = i)$. The probability distribution of the next time step $t+1$ is given by $\pi(t+1)^T = \pi(t)^T P$. With simple induction argument the distribution at time t is given by

$$\pi(t)^T = \pi(0)^T P^t$$

By looking at a discrete-time Markov chain of the graph $\mathcal{G}$, its transition matrix P must satisfy some constraints, i.e,

$$P \geq 0, \ P\mathbf{1} = \mathbf{1}, \ P = P^T \tag{23}$$

where the inequality $P \geq 0$ means that elementwise, so $P_{ij} \geq 0$ for $i, j = 1, \ldots, n$ and $\mathbf{1}$ is a vector where its elements are all one. $P\mathbf{1} = \mathbf{1}$ means that the sum of each row is one. The last equality $P = P^T$ is that P is symmetric, i.e, $P_{ij} = P_{ji}$ for $i, j = 1, \ldots, n$. Since P is symmetric, the column sum of P is also 1. A matrix P that satisfy the the conditions in equation 23 is called *doubly stochastic*. In addition, it must also satisfy

$$P_{ij} = 0, \ (i, j) \notin \mathcal{E} \tag{24}$$

which means that transition can only be allowed if the two vertices is connected by an edge. In article Boyd, Diaconis, and Xiao [2], focuses on Markov chains which are both irreducible and aperiodic, which we will also do. For such chains, we mentioned theorem 2.3, in chapter 2, that says, if a chain is irreducible and aperiodic, then the distribution $\pi(t)$ converges to the unique equilibrium distribution as t becomes large (see Levin, Peres, and Wilmer [8] for proof). The uniform distribtution $(1/n)\mathbf{1}$ is an equilibrium distribution for the Markov chain, and to see this, we have that

$$(1/n)\mathbf{1}^T P = (1/n)\mathbf{1}^T P^T = (1/n)(P\mathbf{1})^T = (1/n)\mathbf{1}^T. \tag{25}$$

The first equality in equation 25 follows from symmetry of P, and the second equality follows from $P\mathbf{1} = \mathbf{1}$.

The rate of the convergence of $\pi(t)$ to the uniform distribution, is determined by the probability matrix P. This is our concern, given a graph $\mathcal{G}$, determine the transition probability matrix P which optimizes the mixing time. Since P is real and symmetric, its eigenvalues are real, and that the magnitudes are less or equal to 1. The latter property is derived from the Perron-Frobenius theory. Let us denote the eigenvalues in nonincreasing order:

$$1 = \lambda_1(P) \geq \lambda_2(P) \geq \ldots \geq \lambda_n(P) \geq -1$$

We can show that this hold by using Gerschgorin circles (see Meyer [10]). The second largest eigenvalue in modulus (shorten as SLEM) of P, $\mu(P)$, is important for the asymptotic rate of convergence of the Markov chain to the uniform equilibrium distribution. It can be formulated as

$$\mu(P) = \max_{i=2,\ldots,n} |\lambda_i(P)| = \max\{\lambda_2(P), -\lambda_n(P))$$

The optimization criterion is that we would like to minimize the SLEM of a transition probability matrix P to get fast mixing. The bounds of the convergence can be measured in many ways, but in [2], one of the bounds is the total variation between two distributions ν and $\tilde{\nu}$ on $\mathcal{V}$. It is defined as the maximum difference in probability assigned to any subset, i.e.,

$$\|\nu - \tilde{\nu}\|_{\text{tv}} = \max_{\mathcal{S}\subseteq\mathcal{V}} \left| \operatorname*{Prob}_{\nu}(S) - \operatorname*{Prob}_{\tilde{\nu}}(S) \right| = (1/2) \sum_i |\nu_i - \tilde{\nu}|$$

The bound on the total variation distance between $\pi(t)$ and the uniform distribution is

$$\sup_{\pi(0)} \|\pi(t) - (1/n)\mathbf{1}\|_{\text{tv}} = (1/2) \max_i \sum_i \left| P_{ij}^t - (1/n) \right| \leq (1/2)\sqrt{n}\mu^t$$

If the Markov chain is irreducible and aperiodic, then $\mu(P) < 1$ and the distribution converges to uniform asymptotically as μ^t. The *mixing rate* is defined as $\log(1/\mu)$ and the *mixing time* is $\tau = 1/\log(1/\mu)$. The mixing time τ gives an asymptotic measure of the required number of steps for the total variation distance of the distribution from uniform to be reduced by the factor e. The mixing rate $\log(1/\mu)$ is approximately $1 - \mu$ when the SLEM is very close to 1. We can look at the mixing rate, mixing time, and the spectral gap as measures for fast mixing. For the setup for the fastest mixing Markov chain problem, we want to find the transition probability matrix that gives the fastest mixing chain. In other words, we would like to

assign the edges of the graph a transition probability such that the SLEM is minimized. To write that out, the problem can be posed as

$$\begin{array}{ll} \text{minimize} & \mu(P) \\ \text{subject to} & P \geq 0,\ P\mathbf{1} = \mathbf{1},\ P = P^T, \\ & P_{ij} = 0,\ (i,j) \notin \mathcal{E} \end{array} \tag{26}$$

P is the optimization variable, and the graph is the problem data. This problem is what we are going to call the fastest mixing Markov chain (FMMC) problem. An optimal SLEM is denoted by μ^* which is given by

$$\mu^* = \inf\{\mu(P) \mid P \geq 0,\ P\mathbf{1} = \mathbf{1},\ P = P^T,\ P_{ij} = 0, (i,j) \notin \mathcal{E}\}$$

There is at least one optimal transition matrix P^*, that is, one for which $\mu(P^*) = \mu^*$. The reason is that μ is continuous and the set of possible transition matrices is compact.

Now that we have formulated the FMMC problem, we will take a look at an example.

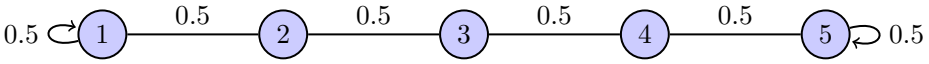

FIGURE 1. A path with 5 vertices. The optimal transition probabilities are shown on the edges of the graph.

EXAMPLE 1.1. *Consider a undirected graph $\mathcal{G}$ with vertex set $\mathcal{V} = \{1, \ldots, 5\}$ and edge set $\mathcal{E} = \{(1,2),(2,3),(3,4),(4,5)\}$ and on each vertex there is an edge to itself (see figure 1). From [3], the paper states that when the graph is a path, the optimal transition probability matrix is given by assigning $1/2$ to all the edges, except self-edges of the vertices $v = 2, \ldots, 4$. The optimal transition probability matrix is*

$$P^* = \begin{pmatrix} 0.5 & 0.5 & 0 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 0 \\ 0 & 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0.5 & 0.5 \end{pmatrix}.$$

From [3], the SLEM for the FMMC problem on a path is given by $\mu^* = \cos(\pi/n) \approx 0.81$, the mixing rate is $\log(1/\mu) \approx 0.21$, the mixining time $\tau = 1/\log(1/\mu) \approx 4.72$ and the spectral gap is $1 - \mu \approx 0.19$.

The article [2] proposes two simple heuristic methods to obtain transition probabilities that give fast mixing. Sometimes it can also be the fastest possible. The first method we are going to look at is called the *maximum-degree chain*, which is a method of assigning the probability based on the degree of the vertex and the maximum vertex degree of the graph. We recall that the degree d_i of vertex i, not counting the self-loop, that is, the number of neighbor vertices of vertex i, not counting itself. The maximum degree of the the graph is given by $d_{\max} = \max_{i \in \mathcal{V}} d_i$. Probability $1/d_{\max}$ is assigned to every non-self-loops of the graph, and letting the self-loop be determined

to ensure that the probabilities at each edge sums to 1. So the elements of the maximum-degree transition probability P^{md} is

$$P_{ij}^{\text{md}} = \begin{cases} 1/d_{\max} & (i,j) \in \mathcal{E} \text{ and } i \neq j \\ 1 - d_i/d_{\max} & i = j \\ 0 & (i,j) \notin \mathcal{E} \end{cases} \tag{27}$$

The second method is called *Metropolis-Hastings chain.* It applies the Metropolis-Hasings algorithm to a random walk on a graph, so it modifies the transition probabilities of a simple random walk on a graph given by

$$P_{ij}^{\text{rw}} = \begin{cases} 1/d_i & (i,j) \in \mathcal{E}, i \neq j \\ 0 & \text{otherwise} \end{cases} \tag{28}$$

Set $R_{ij} = (\pi_j P_{ij}^{\text{rw}})/(\pi_i P_{ij}^{\text{rw}})$ where $\pi = (\pi_1, \ldots, \pi_n)$ is the equilibrium distribution. Then we can obtain a reversible Markov chain by modify P_{ij}^{rw} as following:

$$P_{ij}^{\text{mh}} = \begin{cases} P_{ij}^{\text{rw}} \min\{1, R_{ij}\} & (i,j) \in \mathcal{E}, i \neq j \\ P_{ii}^{\text{rw}} + \sum_{(i,k)\in\mathcal{E}} P_{ik}^{\text{rw}}(1 - \min\{1, R_{ik}\}) & i = j \end{cases} \tag{29}$$

If π is the uniform distribution, then the transition probability matrix P^{mh} is symmetric and can be simplified as

$$P_{ij}^{\text{mh}} = \begin{cases} \min\{1/d_i, 1/d_j\} & (i,j) \in \mathcal{E}, i \neq j, \\ \sum_{(i,k)\in\mathcal{E}} \max\{0, 1/d_i - 1/d_k\}) & i = j, \\ 0 & (i,j) \notin \mathcal{E} \end{cases} \tag{30}$$

The transition probability of a Metropolis-Hastings chain is only dependent on the degrees of its two adjacent vertices.

1.1. Formulation of the problem as a convex optimization program. The transition matrix, P, that describes the Markov chain, has to satisfy some constraints. The entry of the matrix P, $P_{i,j}$, is the probability (or the weight) of the edge (i,j). The probability has to be nonnegative, which gives us that $P_{ij} \geq 0$ for $1 \leq i,j \leq n$, so $P \geq 0$. If the edge (i,j) is not in the graph, we set the transition probability to zero. If we look at a row of P, say i, the elements corresponds to the transition probability of getting from i to j for $j = 1, \ldots, n$. Every row of P sums to 1 and the same holds which follows from symmetry. The graph we are looking at is undirected, which means that if $(i,j) \in \mathcal{E}$ is equivalent with $(j,i) \in \mathcal{E}$ and if $(i,j) \notin \mathcal{E} \Leftrightarrow (j,i) \notin \mathcal{E}$. Thus, the transition matrix P is symmetric. All together, we have

$$\begin{aligned} &\text{minimize} && \mu(P) = \|P - (1/n)\mathbf{1}\mathbf{1}^T\|_2 \\ &\text{subject to} && P \geq 0, P\mathbf{1} = \mathbf{1}, P = P^T, \\ & && P_{ij} = 0,\ (i,j) \notin \mathcal{E} \end{aligned} \tag{31}$$

1.2. Formulation of the problem as a semidefinite program. The fastest mixing Markov chain problem can also be viewed as semidefinite program. We have until now seen that the problem can be formulated as a convex optimization problem with linear equalities as constraints and a convex objective function. Boyd, Diaconis, and Xiao [2] shows that equation

31 can be formulated as SDP, so here we will go through the steps in detail by using theory from Boyd and Vandenberghe [6] and linear algebra (see [10, 11]).

The basic idea of doing this, is to set a variable equal to the norm and minimizing that variable. In the semidefinite program, we have added a new variable s, so that the problem has two variables, s and P, where P is the transition probability matrix in equation 31. Set $s = \|P - (1/n)\mathbf{1}\mathbf{1}^T\|_2$. We will add a new constraint to the SDP and by using Rayleigh quotient defined as $R(A, x) = x^T Ax/x^T x$ for $x \in \mathbf{R}^n \setminus \{0\}$ we have $-s \leq x^T(P - (1/n)\mathbf{1}\mathbf{1}^T)x/x^T x \leq s$. Consider the upper bound so we have $x^T(P - (1/n)\mathbf{1}\mathbf{1}^T)x \leq sx^T x = x^T(sI)x$ which means that $x^T(P - (1/n)\mathbf{1}\mathbf{1}^T)x \leq 0$. Hence $P - (1/n)\mathbf{1}\mathbf{1}^T - sI$ is negative semidefinite i.e $P - (1/n)\mathbf{1}\mathbf{1}^T \preceq sI$. Similar procedure can be done for the lower bound. We have $x^T(P - (1/n)\mathbf{1}\mathbf{1}^T)x \geq -sx^T x = x^T(-sI)x$ such that $x^T(sI + P - (1/n)\mathbf{1}\mathbf{1}^T)x \geq 0$. Hence $sI + P - (1/n)\mathbf{1}\mathbf{1}^T$ is positive semidefinite i.e $sI + P - (1/n)\mathbf{1}\mathbf{1}^T \succeq 0$ such that $-sI \preceq P - (1/n)\mathbf{1}\mathbf{1}^T$. By putting together two constraints derived from the upper and lower bound of the eigenvalue, we get the constraint

$$-sI \preceq P - (1/n)\mathbf{1}\mathbf{1}^T \preceq sI.$$

For the semidefinite program we also want the norm as small as possible. We can now minimize over s to obtain the objective function of the semidefinite formulation. The formulation can be expressed as

$$\begin{aligned} &\text{minimize} && s \\ &\text{subject to} && -sI \preceq P - (1/n)11^T \preceq sI, \\ &&& P \geq 0,\ P1 = 1,\ P = P^T, \\ &&& P_{ij} = 0,\ (i,j) \notin \mathcal{E} \end{aligned} \tag{32}$$

1.3. Convexity of SLEM. Boyd, Diaconis, and Xiao [2] takes care of three ways of proving the convexity of the problem, but here we will look at one of them. We are going to show that the SLEM μ is a convex function of P by using theory found in Boyd and Vandenberghe [6]. Here we will emphasize the steps in further detail than it is shown in the article to proove the convexity of SLEM.

In [2], it is stated that the SLEM is given by

$$\begin{aligned} \mu(P) &= \|(I - (1/n)\mathbf{1}\mathbf{1}^T)P(I - (1/n)\mathbf{1}\mathbf{1}^T\|_2 \\ &= \|P - (1/n)\mathbf{1}\mathbf{1}^T\|_2 \end{aligned} \tag{33}$$

where $\|\cdot\|_2$ is the spectral norm. We will show the first equality in equation 33 this by using lemmas derived from linear algebra (see [12]).

Since P is a symmetric matrix, we know from the spectral theorem that its eigenvectors are orthogonal on each other. Let $\lambda_1, \ldots, \lambda_n$ be the eigenvalues of P and $v_1, \ldots, v_n$ are the corresponding eigenvectors.

LEMMA 1.1. *If $P \in \mathbb{R}^{n\times n}$ be a symmetric matrix and λ_1 is its largest eigenvalue, then*

$$\lambda_1 = \sup_{x\in\mathbb{R}^n, \|x\|=1} x^T Px$$

PROOF. To show the lemma, let λ_1 be the largest eigenvalue of P and the v_1 the corresponding orthonormal eigenvector. Then we have that for $x \in \mathbb{R}^n$

$$\sup_{\|x\|=1} x^T P x \geq v_1^T P v_1 = v_1 \lambda_1 v_1 = \lambda_1 \tag{34}$$

where the last equality follows from that $\|v_1\| = 1$. Since the eigenvectors of P spans $\mathbb{R}^n$ we have that any vector $x \in \mathbb{R}^n$ can be written as a linear combination of the eigenvectors of P, and let $x = \lambda_1 v_1 + \ldots + \alpha_n v_n$ be the vector which maximizes the $x^T P x$. Then we have that

$$\begin{aligned}\sup_{x \in \mathbb{R}^n, \|x\|=1} x^T P x = x^T P x &= (\alpha_1 v_1 + \ldots + \alpha_n v_n)^T P (\alpha_1 v_1 + \ldots + v_n) \\ &= \sum_{i=1}^{n} \alpha_i^2 \lambda_i\end{aligned}$$

The last equality follows from the fact that the eigenvectors are orthogonal on each other, so if $i \neq j$ we have that $v_i P v_j = v_i \lambda_j v_j = \lambda_j v_i v_j = 0$ since $v_i v_j = 0$. Furthermore, we have that $\sum_{i=1}^{n} \alpha_i^2 = \|x\|^2 = 1$. For $x \in \mathbb{R}^n$ we have then that

$$\sup_{\|x\|=1} = \sum_{i=1}^{n} \alpha_i^2 \lambda_i \leq \lambda_1 \sum_{i=1}^{n} \alpha_i^2 = \lambda_1 \tag{35}$$

where the inequality follows from that $\lambda_1 \geq \lambda_i$ for $i = 1, \ldots, n$. This shows that λ_1 is bounded below and above by the sup which yields equality. □

LEMMA 1.2. *If $P \in \mathbb{R}^{n \times n}$ be a symmetric matrix and λ_1 is its largest eigenvalue with corresponding eigenvector v_1, then the second largest eigenvalue is*

$$\lambda_2 = \sup_{x \in \mathbb{R}^n, \|x\|=1} x^T (I - (1/n)\mathbf{1}\mathbf{1}^T) P (I - (1/n)\mathbf{1}\mathbf{1}^T) x \tag{36}$$

PROOF. To show equation 36, we will consider the projection of x onto the subspace $K = \{u \in \mathbb{R}^n : v_1^T u = 0\}$. By linear algebra (for more details see Lay [11]), the projection onto the subspace can be written as $\text{proj}_K(x) = (I - (1/n)\mathbf{1}\mathbf{1}^T)x$. Since the eigenvectors are orthogonal on each other we can use lemma 1.1 to derive the equation for the second largest eigenvalue λ_2 which is

$$\lambda_2 = \sup_{x \in \mathbb{R}^n, \|x\|=1, x \perp v_1} x^T P x \tag{37}$$

It follows that

$$\begin{aligned}\sup_{\|x\|=1, x \perp v_1} x^T P x &= \sup_{\|x\|=1} [(I - (1/n)\mathbf{1}\mathbf{1}^T)x]^T P [(I - (1/n)\mathbf{1}\mathbf{1}^T x] \\ &= \sup_{\|x\|=1} x^T (I - (1/n)\mathbf{1}\mathbf{1}^T) P (I - (1/n)\mathbf{1}\mathbf{1}^T) x\end{aligned}$$

□

LEMMA 1.3. *If $P \in \mathbb{R}^{n\times n}$ be a symmetric matrix and λ_1 is its largest eigenvalue with corresponding eigenvector v_1, then the second largest eigenvalue is*

$$\lambda_n = \inf_{x\in\mathbb{R}^n, \|x\|=1} x^T(I-(1/n)\mathbf{1}\mathbf{1}^T)P(I-(1/n)\mathbf{1}\mathbf{1}^T)x \tag{38}$$

PROOF. The proof of the lemma is similar to the previous lemma, but here we want to take the infimum to obtain the smallest eigenvalue λ_n. □

If we look at the second largest eigenvalue modulus μ of P which is defined as $\mu(P) = \max(\lambda_2, -\lambda_n)$, it is equivalent to looking at the spectral norm of $(I-(1/n)\mathbf{1}\mathbf{1}^T)P(I-(1/n)\mathbf{1}\mathbf{1}^T)$, which is $\mu(P) = \|(I-(1/n)\mathbf{1}\mathbf{1}^T)P(I-(1/n)\mathbf{1}\mathbf{1}^T)\|_2$. Hence we can express the SLEM as a norm since we know from convexity (see Boyd and Vandenberghe [6]) that any norm is a convex function.

1.4. Primal and dual formulation of the FMMC problem. The semidefinite program which is stated in [2] to find the fastest mixing Markov chain by optimizing the second largest eigenvalue in modulus is

$$\begin{array}{ll} \text{minimize} & s \\ \text{subject to} & -sI \preceq P-(1/n)\mathbf{1}\mathbf{1}^T \preceq sI \\ & P \geq 0,\ P\mathbf{1}=\mathbf{1},\ P=P^T, \\ & P_{ij}=0,\ (i,j)\notin\mathcal{E}. \end{array} \tag{39}$$

where the variables are the matrix P and the scalar s. We refer this problem to the primal problem of the FMMC. The related problem, the dual problem, can be stated as

$$\begin{array}{ll} \text{maximize} & \mathbf{1}^T z \\ \text{subject to} & Y\mathbf{1}=0,\ Y=Y^T,\ \|Y\|_* \leq 1, \\ & (z_i+z_j)/2 \leq Y_{ij},\ (i,j)\in\mathcal{E} \end{array} \tag{40}$$

where the variables are $z \in \mathbb{R}^n$ and $Y \in \mathbb{R}^{n\times n}$. The *dual norm* is defined as $\|Y\|_* = \sum_{i=1}^n |\lambda_i(Y)|$, which is the sum of the singular values of Y.

We will now go into details of the relation between the primal and the dual problem by using theory from Boyd and Vandenberghe [6, 13]. To show the transformation from primal to dual we will consider the Lagrange function of the primal problem and introduce variables, and then optimize the function in order to obtain the dual problem.

First, we introduce new variables for the constraints for the problem. Let A, B, Γ, Λ be $n \times n$ symmetric matrices, and $z \in \mathbb{R}^n$. In addition, $\lambda_{i,j} = \Lambda_{ij} = 0$ for $(i,j) \in \mathcal{E}$ and we require that A, B are positive semidefinite, i.e,

$A, B \succeq 0$. The Lagrange function is

$$\begin{aligned} L(s, P, A, B, \Gamma, z, \Lambda) &= s + \mathrm{Tr}(A(-P + (1/n)\mathbf{1}\mathbf{1}^T - sI)) + \\ &\quad \mathrm{Tr}(B(P - (1/n)\mathbf{1}\mathbf{1}^T - sI)) - \\ &\quad \mathrm{Tr}(\Gamma P) + z^T(1 - P1) + \mathrm{Tr}(\Lambda P) \\ &= s(1 - \mathrm{Tr}(A + B)) + \\ &\quad \mathrm{Tr}(P(B - A - \Gamma + \Lambda - (1/2)(\mathbf{1}z^T + z\mathbf{1}^T)) + \\ &\quad \mathbf{1}^T z + \mathrm{Tr}((1/n)\mathbf{1}^T(A - B)\mathbf{1}) \end{aligned} \tag{41}$$

Using the fact that P is symmetric, we get that $z^T P\mathbf{1} = (1/2)\,\mathrm{Tr}(\mathbf{1}z^T + z\mathbf{1}^T)$. The dual is the infimum of the Lagrange function with respect to its variables, so if we minimize L over s and P we will get

$$1 = \mathrm{Tr}(A + B) \text{ and } B - A - \Gamma + \Lambda = (1/2)(\mathbf{1}z^T + z\mathbf{1}^T). \tag{42}$$

All entries of Γ are non-negative, $\Gamma \geq 0$.

$$(1/2)(\mathbf{1}z^T + z\mathbf{1}^T) = B - A - \Gamma + \Lambda \leq B - A + \Lambda \tag{43}$$

For $(i, j) \in \mathcal{E}$ then $\lambda_{i,j} = 0$ which means that

$$(1/2)(z_i + z_j) \leq B_{ij} - A_{ij} + \lambda_{ij} = B_{ij} - A_{ij} \tag{44}$$

If we now put everything together we can formulate the dual as

$$\begin{array}{ll} \text{maximize} & \mathbf{1}^T z - \mathrm{Tr}((1/n)\mathbf{1}^T(B - A)\mathbf{1}) \\ \text{subject to} & \mathrm{Tr}(A + B) = 1 \\ & (1/2)(z_i + z_j) \leq (B - A)_{ij}, \text{ if } (i, j) \in \mathcal{E} \\ & A, B \succeq 0 \end{array} \tag{45}$$

Define $Y = B - A$, and note that Y is symmetric since A, B are symmetric, so we have $Y = Y^T$. The diagonal elements of the matrices, A and B, are nonegative, hence they are positive semidefinite. By this property, we have that

$$\|B\|_* + \|A\|_* = \sum_{i=1}^{n} |\lambda_i(B)| + \sum_{i=1}^{n} |\lambda_i(A)| \tag{46}$$

$$= \sum_{i=1}^{n} \lambda_i(B) + \sum_{i=1}^{n} \lambda_i(A) \tag{47}$$

$$= \mathrm{Tr}(B) + \mathrm{Tr}(A) = 1 \tag{48}$$

The first equality follows from definition of the dual norm. Next, since A and B are positive semidefinite and real, we know that their eigenvalues are positive, that is, $\lambda_i(A), \lambda_i(B) \geq 0 \; \forall \; i = 1, \ldots, n$. The trace of A and B can be defined as a sum of their eigenvalues which gives the third equality. Last, we obtain the last equality from the first formulation of the dual problem that the sum of the traces of A and B is 1. So far, we have obtain that $\|B\|_* + \|A\|_* = 1$.

$$\|Y\|_* = \|B - A\|_* \leq \|B\|_* + \|A\|_* = 1 \tag{49}$$

From [6], we have that $(B-A)\mathbf{1} = Y\mathbf{1} = 0$ as we have

$$\begin{aligned} g(A,B,\Gamma,z,\Lambda) &= \inf_{s,P} L(s,P,A,B,\Gamma,z,\Lambda) \\ &= \mathbf{1}^T z - \mathrm{Tr}((1/n)\mathbf{1}^T(B-A)\mathbf{1}) \\ &= \begin{cases} \mathbf{1}^T z, & \mathrm{Tr}((1/n)\mathbf{1}^T(B-A)\mathbf{1}) = 0 \\ -\infty, & \text{otherwise} \end{cases} \end{aligned}$$

By the equation above we get that $g(A,B,\Gamma,z,\Lambda) = \mathbf{1}^T z$ if $\mathrm{Tr}((1/n)\mathbf{1}^T(B-A)\mathbf{1}) = 0$ which implies that $(B-A)\mathbf{1} = Y\mathbf{1} = 0$. Furthermore, we have that $\mu^* \geq g(A,B,\Gamma,z,\Lambda)$, so by maximizing $\mathbf{1}^T z$ when $Y\mathbf{1} = 0$, $Y = Y^T$ and the constraints in equation 45, we will get the equation 40.

2. Applications of the fastest mixing Markov chain

We will in the following section motivate the fastest mixing Markov chain problem by looking at two applications. The applications of the problem we are going to look at are card shuffle and cup shuffle. A question we may ask for such problems are for how long do we have to shuffle before we obtain randomness?

2.1. Card Shuffle. Let us consider a deck of n cards, where the cards are labeled with integers from 1 to n. A *permutation* of the deck is a way of ordering the cards. For instance, a natural ordering of a deck of n cards can be denoted by $1 \cdots n$. There are many ways of shuffle a deck of cards. A shuffle is a method of arranging the order of the cards. For a deck of n cards we have $n!$ permuations. We will denote S_n as the set of all permutations with a deck of n cards. A permuation $x \in S_n$ can be denoted as a veector of n entries with distinct numbers. For example, a permuation x of 5 cards, where the top card at position 0 is 3 and the bottom card at position $n-1$ is 4 is given by $x = (3,2,1,5,1)$.

The idea behind the application of the FMMC problem for card shuffling is that we can represent the suffling method as a graph. The vertices of the graph corresponds to all the permuations of the shuffle and each edge is a possible transition that we can get from a permuation to permuation. The intention behind a shuffle is to achieve randomness, for example when we pick a card from the deck the probability of picking any card should be the same for all the cards. A shuffle can therefore be described as a random walk on a graph. A simulation of a random walk can be found in appendix B. In addition, we want to use as short time on the shuffle which means that we want fast convergence of the random walk to the equilibrium distribution. Of course, when we shuffle it is preferable that all permutations can be reached within a time, so the graph has to be connected.

Example 2.1 (Card shuffle with 3 cards). *We will now look at an example of a deck of* 3 *cards. The permutation set* S_3 *is given by* $S_3 = \{x_1, x_2, \ldots, x_6\}$ *where* $x_1 = (1,2,3)$, $x_2 = (1,3,2)$, $x_3 = (2,1,3)$, $x_4 = (2,3,1)$, $x_5 = (3,1,2)$ *and* $x_6 = (3,2,1)$. *We will now consider a shuffle method* Q *in order to determine the graph. Let say* Q *denotes the shuffle method where we move the top card to one of the* n *positions or, take the card at position* n *and place it at the top. The edges of the graph are*

(x_1, x_4),(x_1, x_3),(x_2, x_6),(x_2, x_5),(x_3, x_2), (x_4, x_5),(x_4, x_6),(x_5, x_1),(x_6, x_3) *including the self-loop at each vertex. Figure 2 shows the graph for the shuffle method.*

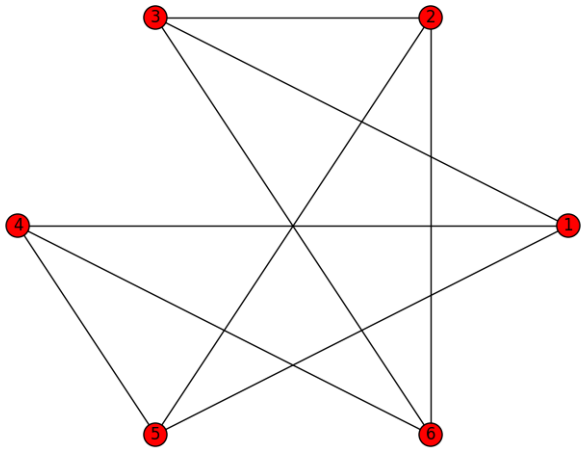

FIGURE 2. The graph representing the shuffle of a deck with 3 cards described in example 2.1.

In example 2.1, we have shown that a shuffle method can be represented as a graph. A shuffle can be consiered as a random walk on a graph. By solving the FMMC problem on a graph, we may be able to answer question as how fast can we achieve a properly mixed deck of cards?

2.2. Cup Shuffle. We can describe a cup shuffle as following: Let us consider n cups flipped upside down on a table. We place a ball under one of the cups, and shuffle. The idea of this is that we want to shuffle the cups so many times such that after a while, the probability that the ball is hidden under one of n cups is equally distributed. A move in this context means that we move the interchange the cup containing the ball with the empty cup which does not. Each vertex of the graph represent the position the ball can be in. An edge that connects two vertices i and j, then the ball can be moved from vertex i to vertex j, or conversely, from vertex j to vertex i. By this, the graph really represents the rule for how we are allowed to move the cup with the ball in different positions. For simplicity, we will assume that we are always able to remain in the same position and the ball can be moved to any position (not necessary in one step). We can therefore associate the a cup shuffle as a random walk on a undirected, connected graph.

EXAMPLE 2.2 (Cup shuffle with 5 cups). *Let us assume that we have* 5 *cups denoted with the positions* 1 *to* 5*. Let say that we are able to move the ball to any position of that we want. So the graph that represents the moving rule of the ball is a graph where all pairs of vertices are connected with an edge. We will also assume that the every vertex has a self-loop, that*

is, we are allowed to not move the ball. Define the vertex set of the graph as $\mathcal{V} = \{1, 2, 3, 4, 5\}$. *The edge set can be denoted by* $\mathcal{E}$ *and contains all pair of vertices, self-loop inclusive. Figure 3 shows the rules of how the ball can moved to all position.*

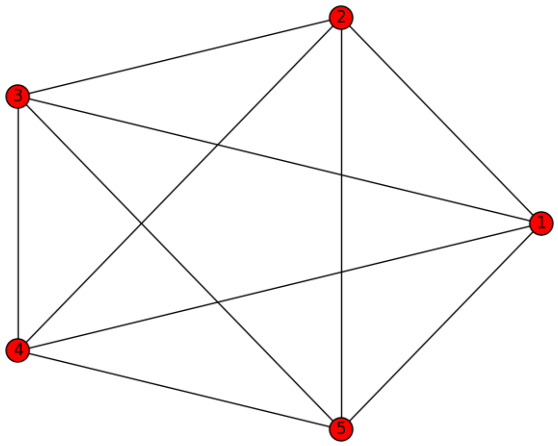

FIGURE 3. The graph represents the moving rule of a 5 cup shuffle.

If we look at the fastest mixing Markov chain on the graph, we will in the chapter derive that the transition probability matrix $P^* \in \mathbb{R}^{n \times n}$ *is given by* $P^* = (1/n)\mathbf{1}\mathbf{1}^T$. *Although, this is the optimal shuffling method, what if we make some restriction on the rules of moving? For instance, a naive approach is to say that we we are only allowed to move the ball one step to the left, to the right or remain in the position. Then the graph that represents the moving rule of the ball is a path with* 5 *vertices. A random walk on the graph represents a shuffling of the cups, so if we consider the optimal shuffle, how fast can mix the cups in order to get uniform probability that the ball is at a certain position after some time?*

CHAPTER 4

Solve the FMMC problem on graphs

This chapter will solve the FMMC problem on different types of graphs using the subgradient method. First, we will look at the problem on the four small graphs from Boyd, Diaconis, and Xiao [2], and then solve the problem on graphs such as paths, cycles and star graphs.

For motivation, we will look at one of the simplest Markov chain on a graph with, namely when the graph has $n = 2$ vertices, which is also taken care of in Boyd et al. [3]. The graph is undirected, connected and the two vertices have self-loop. Since we do not require that $P_{ij} = 0$ for any edge $(i, j) \notin \mathcal{E}$, we can choose P as we want, but it must satisfy $P = P^T$, $P \geq 0$ and $P\mathbf{1} = \mathbf{1}$. The transition probability matrix is on the form

$$P = \begin{pmatrix} x & 1-x \\ 1-x & x \end{pmatrix}. \tag{50}$$

using that $P\mathbf{1} = \mathbf{1}$ and $P = P^T$. We can find the SLEM value by computating the eigenvalues of P and minimize the second largest eigenvalue. Consider the equation $\det(P - \lambda I) = 0$, and solve for λ, which gives $(x - \lambda)^2 - (1 - x)^2 = \lambda^2 - 2x\lambda + (2x - 1) = 0$. The eigenvalues of P are 1 and $2x-1$. From the constraint $P \geq 0$, we have that $x \in [0, 1]$, but we wanted to minimize the second largest eigenvalue modulus, so we obtain the smallest if $x = 1/2$.

Another way to solve the FMMC problem for the given graph, is to use the definition of SLEM from equation 33, where it was expressed as a spectral norm of $P - (1/n)\mathbf{1}\mathbf{1}^T$. In this way, we see that by choosing $P = (1/2)\mathbf{1}\mathbf{1}^T$ we have found the optimal transition probability matrix with $\mu = 0$ as optimal SLEM value.

LEMMA 0.1. *If graph $\mathcal{G}$ is undirected and connected such that every pair of vertices are connected with an edge and every vertex have a self-loop, then the optimal transition probability matrix is $P^* = (1/n)\mathbf{1}\mathbf{1}^T$ where n is the number of vertices in the graph.*

PROOF. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote the graph. Then we have that that all edges $(i, j) \in \mathcal{E}$, which means that the contraint $P_{ij} = 0$ for $(i, j) \notin \mathcal{E}$ can be discarded, since all edges $(i, j) \in \mathcal{E}$. The FMMC problem reduces to finding a doubly stochastic matrix which minimizes the SLEM. Suppose $P = (1/n)\mathbf{1}\mathbf{1}^T$. Then we have from equation 33 that we can denote the SLEM with use of spectral norm, so we have that

$$\mu^* = \|P - (1/n)\mathbf{1}\mathbf{1}^T\|_2 = 0$$

which means that it is the optimal transition matrix for the FMMC problem on $\mathcal{G}$. □

In this section we will look at Markov chain simulations and see the progress on different types of graphs. We will start by considering the four small graphs from [3]. Next, we will continue by looking at simple graphs such as a path, a cycle and a star graph, and use the fastest mixing Markov chain that we can find to compute the progrss of the convergence of the probability distribution for some time steps.

We recall from Markov chain in chapter 2, that the the probability distribution at time step t can be computed as $\pi(t) = \pi(0)P^t$ for $t \geq 0$. To measure the distance between the distribution $\pi(t)$ at time t and the equilibrium distribution $(1/n)\mathbf{1}$, we use the total variation distance from equation 1. For simulation we can consider this this implementation:

```
import numpy as np

def sim(u0, P, time):
        """
        Simulation of a Markov chain for
        Attributes:
        u0 - initial probability distribution
        P - transition probabability distribution
        time - number of time steps

        Return:
        u - probability distributions for the time steps
        """
        n = len(P)
        u = np.zeros((time+1,n))
        u[0] = u0
        t = 0
        while t < time:
                u[t+1] = np.dot(u[t],P)
                t += 1
        return u
```

1. Fastest mixing Markov chain on small graphs

In this part we will show the optimal solution for some small graphs along with the mixing rate, mixing time and spectral gap. We will use the subgradient method that is described in Boyd, Diaconis, and Xiao [2] to solve the FMMC problem on various types of graphs. First, we will solve for the four small graphs described in [2]. Then we will solve and analyze the solutions for the problem on paths, cycles and star graphs.

1.1. Four small graphs. Some numerical examples are given for the FMMC problems on small graphs in [2]. We will here try to verify the results in the article (see section 3 in [2]), by solving looking at the same

graphs. The examples that we are going to show compare, the maximum-degree and Metropolis-Hastings chains with the fastest mixing chain. For the purpose of verifying the solution in Boyd, Diaconis, and Xiao [2] we will run the subgradient method on the same graphs and represent the solutions.

The four small graphs we are going to take a closer look at are described in figure 1a, 1b, 1c and 1d.

In table 2, we have solved the FMMC problem for the four small graphs which confirms the result given in [2]. It shows the SLEM values of the Markov chains for each graph, and the transition probability of the fastest mixing chain. For the graph in figure 1a, we see that the SLEM value of the maximum-degree and Metropolis-Hastings chain are the optimal. The SLEMs for graph (b) and (c) are not optimal by the maximum degree chain and the Metropolis-Hastings chain. For graph (d), the maximum-degree chain gives the optimal SLEM value, but the Metropolis-Hastings chain will not. We also note that the optimal transition matrix will not necessary be unique, which can be seen by comparing the transition probability matrix for graph (b) with Table 1 in [2] (see section 3.1).

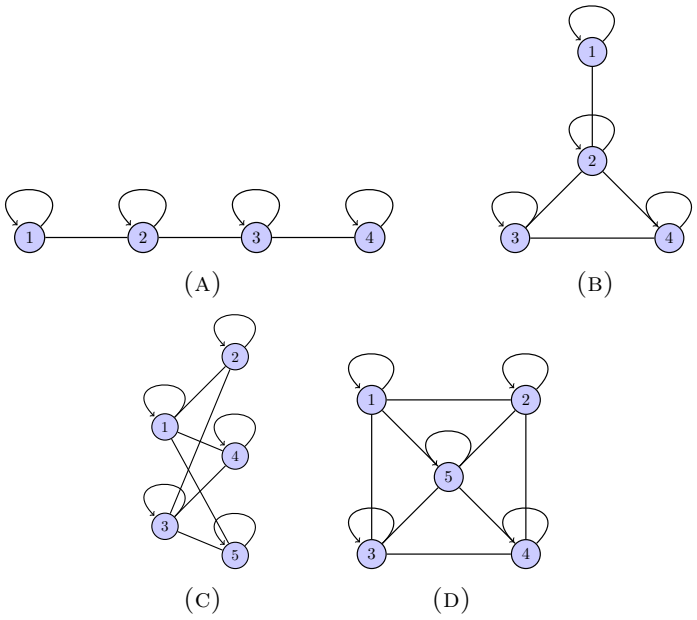

FIGURE 1. Four small graphs

In the next section we will take a closer look at the FMMC problem on a path.

1.2. Path. From Boyd et al. [3], the solution of the FMMC problem on a simple path is a tridiagonal matrix with 0.5 on the non-zero entries. Since we know the optimal transition probability matrix of such problems, we will consider two transition probability matrices P_1 and P_2 on a path and simulate the convergence of the matrices. Doing this, we hope to compare the rate of convergence for the transition matrices.

Graph	μ^{md}	μ^{mh}	μ^*
(A)	0.707	0.707	0.707
(B)	0.667	0.667	0.636
(C)	0.667	0.667	0.429
(D)	0.250	0.583	0.250

TABLE 1. The table shows the SLEM values for the maximum-degree and Metropolis-Hastings chain, and the optimal SLEM values on the four small graphs in figure 1.

Graph	Optimal transition matrix P^*
(a)	$\begin{pmatrix} 0.500 & 0.500 & 0.000 & 0.000 \\ 0.500 & 0.000 & 0.500 & 0.000 \\ 0.000 & 0.500 & 0.000 & 0.500 \\ 0.000 & 0.000 & 0.500 & 0.500 \end{pmatrix}$
(b)	$\begin{pmatrix} 0.545 & 0.455 & 0.000 & 0.000 \\ 0.455 & -0.000 & 0.273 & 0.273 \\ 0.000 & 0.273 & 0.227 & 0.500 \\ 0.000 & 0.273 & 0.500 & 0.227 \end{pmatrix}$
(c)	$\begin{pmatrix} 0.143 & 0.286 & 0.000 & 0.285 & 0.286 \\ 0.286 & 0.428 & 0.286 & 0.000 & 0.000 \\ 0.000 & 0.286 & 0.143 & 0.285 & 0.286 \\ 0.285 & 0.000 & 0.285 & 0.429 & 0.000 \\ 0.286 & 0.000 & 0.286 & 0.000 & 0.429 \end{pmatrix}$
(d)	$\begin{pmatrix} 0.250 & 0.250 & 0.250 & 0.000 & 0.250 \\ 0.250 & 0.250 & 0.000 & 0.250 & 0.250 \\ 0.250 & 0.000 & 0.250 & 0.250 & 0.250 \\ 0.000 & 0.250 & 0.250 & 0.250 & 0.250 \\ 0.250 & 0.250 & 0.250 & 0.250 & 0.000 \end{pmatrix}$

TABLE 2. The optimal transition matrices for the four small graphs.

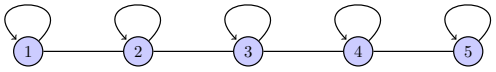

FIGURE 2. A path with 5 vertices. Each vertex has an edge to itself and edge to its neighbor.

Let us consider a path with 5 vertices (see figure 2). [3] states that the optimal transition probability matrix for the FMMC of a path is given by assigning the transition probability 0.5 to every edge in the graph, except for the self-loop on the vertices which are not at the ends (first and last vertex). In other words, optimal transition probability matrix is a tridiagonal matrix where each non-zero entry is 0.5.

The optimal transition probability matrix can be written as

$$P^* = \begin{pmatrix} 0.5 & 0.5 & 0 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 0 \\ 0 & 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0.5 & 0.5 \end{pmatrix}$$

We will compare the optimal transition probability matrix with other transition probability matrices on a path for the FMMC problem and make a simulation of the distribution when we start at vertex 0. Let the probability distribution vector $u \in \mathbb{R}^n$ be defined as $u = (1, 0, \ldots, 0)$.

$$P_1 = \begin{pmatrix} 0.3 & 0.7 & 0 & 0 & 0 \\ 0.7 & 0 & 0.3 & 0 & 0 \\ 0 & 0.3 & 0 & 0.7 & 0 \\ 0 & 0 & 0.7 & 0 & 0.3 \\ 0 & 0 & 0 & 0.3 & 0.7 \end{pmatrix} \text{ and } P_2 = \begin{pmatrix} 0.4 & 0.6 & 0 & 0 & 0 \\ 0.6 & 0.1 & 0.3 & 0 & 0 \\ 0 & 0.3 & 0.2 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0.4 & 0.1 \\ 0 & 0 & 0 & 0.1 & 0.9 \end{pmatrix}$$

Figure 3 shows the convergence of the distribution when we start at vertex 0 when we apply it to the transition probability matrices P_{opt}, P_1 and P_2 for the time interval $[0, 30]$. For each iterations we observe that the distance reduces asymptotic to zero, and that the optimal solution converges fastest. P_2 converges slowest of the three transition probability matrices, although, it matches the other two, at the first five time iterations.

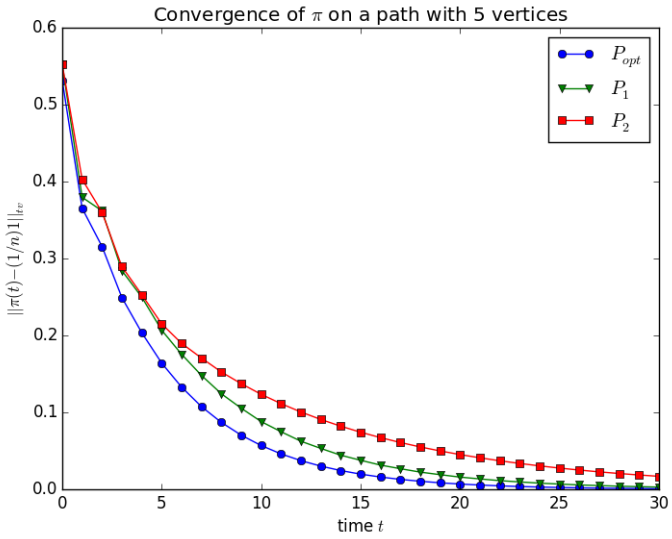

FIGURE 3. The plot shows the convergence of three Markov chains on a path graph with 5 vertices. The transition probability matrices of the chains are P_{opt}, P_1 and P_2. The initial probability distribution $\pi(0)$ is randomly generated. And we use the total variation distance to measure the distance between the two distributions $\pi(t)$ and $(1/n)\mathbf{1}$ at a time t.

When we are using the subgradient method to find the optimal solution for a simple path, we do not have to do many iteration. Actually, the optimal solution is obtained initially by applying either the maximum degree chain method or the Metropolis-Hasting chain method for uniform distribution. This follows directly as the subgradient method needs a starting point to modify the solution. But since the solution we have is the optimal, all other iterations will be discarded.

Table 3 shows the SLEM value, mixing rate, mixing time and the spectral gap for the three transition matrices defined above. We see that P^* has the smallest SLEM value μ compared to the rest, and therefore it will converge faster. We also see that the mixing time for P^* is half the size as the mixing time for the P_2.

	SLEM μ	Mixing rate $\log(1/\mu)$	Mixing time $\tau = 1/\log(1/\mu)$	Spectral gap $1-\mu$
P^*	0.809	0.212	4.718	0.788
P_1	0.842	0.171	5.834	0.829
P_2	0.906	0.098	10.175	0.902

TABLE 3. Comparison of SLEM, mixing rate, mixing time and the spectral gap for the transition probability matrices P^*, P_1 and P_2.

1.3. Cycle graph. In this section we will compute the optimal transtition probability matrix of the FMMC problem on a cycle. We will make the same analysis as in the previous section to see what these optimal transition probability matrices look like.

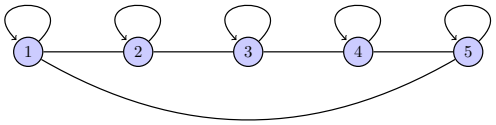

FIGURE 4. Cycle graph with 5 vertices.

Consider a 5-cycle graph (as described in figure 4), a cycle with 5 vertices. When we compute optimal solution using the subgradient method we get that the optimal SLEM value is achieved with the transition probability matrix given by

$$P^* = \begin{pmatrix} 0.2 & 0.4 & 0 & 0 & 0.4 \\ 0.4 & 0.2 & 0.4 & 0 & 0 \\ 0 & 0.4 & 0.2 & 0.4 & 0 \\ 0 & 0 & 0.4 & 0.2 & 0.4 \\ 0.4 & 0 & 0 & 0.4 & 0.2 \end{pmatrix}$$

We will also consider two transition probability matrices P_1 and P_2 on the 5-cycle graph given by

$$P_1 = \begin{pmatrix} 0.1 & 0.2 & 0 & 0 & 0.7 \\ 0.2 & 0.1 & 0.7 & 0 & 0 \\ 0 & 0.7 & 0.1 & 0.2 & 0 \\ 0 & 0 & 0.2 & 0.6 & 0.2 \\ 0.7 & 0 & 0 & 0.2 & 0.1 \end{pmatrix}$$

$$P_2 = \begin{pmatrix} 0.1 & 0.1 & 0 & 0 & 0.8 \\ 0.1 & 0.5 & 0.4 & 0 & 0 \\ 0 & 0.4 & 0.3 & 0.3 & 0 \\ 0 & 0 & 0.3 & 0.5 & 0.2 \\ 0.8 & 0 & 0 & 0.2 & 0 \end{pmatrix}$$

Figure 5 shows the convergence of the probability distribution between $\pi(t)$ at a time t to the uniform equilibrium distribution $(1/n)\mathbf{1}$ on a cycle with 5 vertices for three Markov chains. The initial probability distribution $\pi(0)$ is randomly generated using the implementation in A.5. We have used the total variation distance two measure the distance between the two distribution and simulated the Markov chain for 20 timesteps. The fastest mixing chain is obtained with P_{opt} as the transition probability matrix.

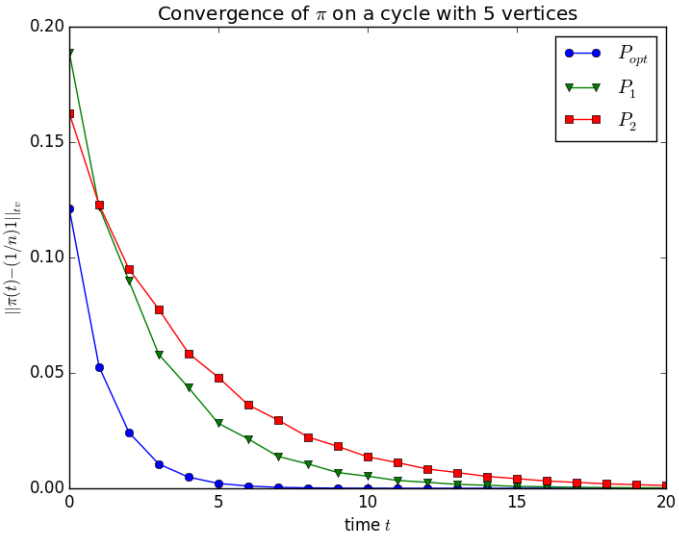

FIGURE 5. The plot shows the convergence of three Markov chains on a 5-cycle graph with the transition probability matrices P_{opt}, P_1 and P_2. The initial probability distribution $\pi(0)$ is randomly generated. We use the total variation distance to measure the distance between the two distributions $\pi(t)$ and $(1/n)\mathbf{1}$ at a time t.

Figure 6 illustrates the transition probability matrix P^* as a graf. For each edge $e = (i, j) \in \mathcal{E}$ where $i \neq j$ we have that its transition probability $P_{ij} = 2/5$. And if $e = (i, j) \in \mathcal{E}$ where $i = j$ then $P_{ii} = 1/5$.

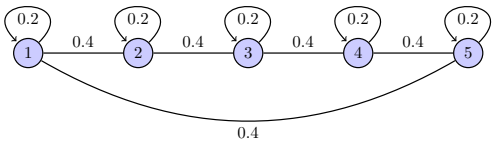

FIGURE 6. A cycle with 5 vertices with the optimial transition probabilities on the edges.

	SLEM μ	Mixing rate $\log(1/\mu)$	Mixing time $\tau = 1/\log(1/\mu)$	Spectral gap $1-\mu$
P^*	0.447	0.805	1.243	0.553
P_1	0.707	0.347	2.885	0.293
P_2	0.783	0.245	4.087	0.217

TABLE 4. Mixing measures for the matrix P_{cycle}.

Since the graph is a cycle, each vertex has two neighbors and the probability of moving one of its neighbor is given by β and the staying probability is α. Generally, we can form the optimal transition probability matrix by the following: Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote the cycle graph. Then the entries of the optimal transition probability matrix P is given by

$$P_{ij} = \begin{cases} \alpha & i = j \\ \beta & (i,j) \in \mathcal{E} \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

where $\alpha, \beta \in \mathbb{R}$ satisfy $\alpha, \beta \geq 0$ and $\alpha + 2\beta = 1$, so the form of the optimal transition matrix is on the form

$$P = \begin{pmatrix} \alpha & \beta & & & \beta \\ \beta & \alpha & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \alpha & \beta \\ \beta & & & \beta & \alpha \end{pmatrix} \tag{51}$$

Number of edges	Time (in seconds)	μ
4	0.01	0.333333
5	0.01	0.447214
6	0.01	0.600000
7	0.00	0.669362
9	0.05	0.784735
10	0.05	0.825665
11	0.42	0.850115
12	0.42	0.874437

TABLE 5. The table shows the SLEM values for the FMMC problem on cycle graphs using the subgradient method.

1.4. Star graph. Boyd et al. [3] shows that the FMMC problem on a path gives a simple form of the optimal transition probability matrix a tridiagonal matrix. In fact, the result of the problem on a star graph has a simple solution too. We will in this section look at the transition probability matrix of the FMMC problem on a star graph.
A nice result for star graphs can be found in [14], which we now will take a look on.

THEOREM 1.1. *If $\mathcal{G}$ is a star graph with n vertices. Then the second largest eigenvalue modulus μi of the FMMC problem is given by $\mu = (n-2)/(n-1)$.*

PROOF. Let $\mathcal{G}$ be a star graph of n vertices. We will assume that the vertices of the graph are similar. Furthermore, we will also assume that the transition probabilities on the edges which are connected to the center vertex are similar, and that the self-loop on vertex j for $j \in \{2, \dots, n\}$ are similar. By the assumptions, we suppose that the optimal transition probability matrix $P \in \mathbb{R}^{n\times n}$ is given by

$$P = \begin{pmatrix} x & z & \cdots & z \\ z & y & & \\ \vdots & & \ddots & \\ z & & & y \end{pmatrix}$$

where only the non-zero elements are given, all other entries not specified are zero. The transition probability matrix form a family of transition probability matrices by determine the entries x, y and z in respect to that it must satisfy $P\mathbf{1} = \mathbf{1}$, $P \leq 0$ and $P = P^T$. We will consider the SLEM value of P by looking at the spectrum of P, that is, the eigenvalues of P. We will find these by computing $\det(P - \lambda I)$ where $\det(A)$ denotes the determinant of a matrix A, and $I \in \mathbb{R}^{n\times n}$ is the identity matrix. Furthermore, we are going to solve for λ by considering the equation

$$\det(P - \lambda I) = 0.$$

We have

$$\begin{aligned} \det(P - \lambda I) &= \begin{vmatrix} x-\lambda & z & \cdots & z \\ z & y-\lambda & & \\ \vdots & & \ddots & \\ z & & & y-\lambda \end{vmatrix} \\ &= (x-\lambda)\det(M_1) + \sum_{k=2}^{n} (-1)^{k-1} z \det(M_k) \end{aligned}$$

where $M_1, \dots, M_n \in \mathbb{R}^{(n-1)\times(n-1)}$ defined by $M_1 = \operatorname{diag}(\underbrace{(y-\lambda, \dots, y-\lambda)}_{n-1})$ and $(M_k)_{ij} = \begin{cases} z & j = 1 \\ y-\lambda & j = i,\ k \leq j \\ y-\lambda & j = i+1,\ j < k \\ 0 & \text{otherwise} \end{cases}$ for $2 \leq k \leq n$. The determinant

$\det M_1 = (y-\lambda)^{n-1}$ and $\det(M_k)$ can be found by consider the minor when we remove the k-th column and $k-1$ row from M_k which gives

$$\begin{aligned}\det(M_k) &= (-1)^k z \det(\operatorname{diag}((y-\lambda,\dots,y-\lambda)) \\ &= (-1)^k z (y-\lambda)^{n-2}\end{aligned}$$

The determinant of $P-\lambda I$ can therefore be written as

$$\begin{aligned}\det(P-\lambda I) &= (x-\lambda)(y-\lambda)^{n-1} + \sum_{k=2}^{n} (-1)^{2k-1} z^2 (y-\lambda)^{n-2} \\ &= (x-\lambda)(y-\lambda)^{n-1} - (n-1)z^2(y-\lambda)^{n-2} \\ &= (y-\lambda)^{n-2}((x-\lambda)(y-\lambda) - (n-1)z^2)\end{aligned}$$

Furthermore, we have that

$$\begin{aligned}(x-\lambda)(y-\lambda) - (n-1)z^2 &= \lambda^2 - (x+y)\lambda + xy - (n-1)z^2 \\ &= \lambda^2 - (2-nz)\lambda + (1-nz)\end{aligned}$$

and by solving the equation $\det(P-\lambda I) = 0$, hence $\lambda^2-(2-nz)\lambda+(1-nz) = 0$ for λ, we get that the solutions are $\lambda_1 = 1$ and $\lambda_2 = 1-nz$. The spectrum of P, $\lambda(P)$, is given by $\lambda(P) = \{1, 1-nz, 1-z\}$. We see that the second larges eigenvalue is dependent on the z which corresponds to the transition probabilities for the non-self-loop edges of the star graph. Since $P \geq 0$ and $P\mathbf{1} = \mathbf{1}$, $x = 1-(n-1)z$ and $y = 1-z$, such that for $n > 1$, we have that $0 \leq z \leq \min(1, 1/(n-1)) = 1/(n-1)$ as $x, y \geq 0$. By choosing $z = 1/(n-1)$ we obtain the smallest eigenvalues for P of the specific family of transition probability matrix. The optimal transition probability matrix have eigenvalues given by $1, -1/(n-1), (n-2)/(n-1)$ where the multiplicities are $n-2$,1 and 1, respectively. The SLEM $\mu(P)$ is

$$\mu(P) = \max\left\{\left|-\frac{1}{n-1}\right|, \left|\frac{n-2}{n-1}\right|\right\} = \frac{n-2}{n-1} \tag{52}$$

This shows that the optimal transition probability matrix on a star graph of n vertices is determined by setting $x = 0$, $y = (n-2)/(n-1)$ and $z = (n-2)/(n-1)$. We will lok □

Consider a star graph which has n vertices.

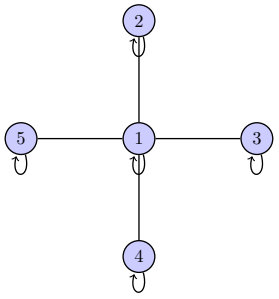

FIGURE 7. A star graph with 5 vertices.

The optimal SLEM value of the FMMC problem is given by

$$\mu^* = \frac{n-2}{n-1}$$

where n is the number of vertices of the star graph ($n > 2$). The optimal transition probability matrix can be written in general form as

$$P_{ij}^* = \begin{cases} \frac{n-2}{n-1} & \text{if } i = j \text{ is not the center vertex} \\ \frac{1}{n-1} & \text{if } (i,j) \in \mathcal{E} \\ 0 & \text{otherwise} \end{cases}$$

Table 6 shows the SLEM values of the optimal transition probability matrices on a star graphs with various number of vertices.

Number of vertices	SLEM μ
3	1/2
4	2/3
5	3/4
6	4/5
7	5/6

TABLE 6. The table shows the SLEM value of the optimal transition probability transition matrix on a star graph.

The starting point when we use the subgradient method, are initialized by the maximum-degree chain or the Metropolis-Hastings chain, to get feasible starting point. However, the optimal solution will be obtained by using the heuristic method, unfortunately, a good stopping criterion is difficult to find, so the subgradient method will go on with the iterations until maximum iterations is reached.

Here is one example of an optimal transition probability matrix of the FMMC problem on a star graph with 5 vertices given by

$$P_{\text{star}}^* = \begin{pmatrix} 0 & 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.75 & 0 & 0 & 0 \\ 0.25 & 0 & 0.75 & 0 & 0 \\ 0.25 & 0 & 0 & 0.75 & 0 \\ 0.25 & 0 & 0 & 0 & 0.75 \end{pmatrix}$$

and the corresponding optimal SLEM value μ_{star}^* is 0.75. Both the Metropolis-Hastings chain and the maximum-degree chain are the same and are given by

$$P_{\text{star}}^{\text{mh}} = P_{\text{star}}^{\text{md}} = \begin{pmatrix} 0 & 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.75 & 0 & 0 & 0 \\ 0.25 & 0 & 0.75 & 0 & 0 \\ 0.25 & 0 & 0 & 0.75 & 0 \\ 0.25 & 0 & 0 & 0 & 0.75 \end{pmatrix}$$

so the corresponding SLEM value $\mu_{\text{star}}^{\text{mh}} = \mu_{\text{star}}^{\text{md}} = 0.75 = \mu_{\text{star}}^*$.

The eigenvalues of P^*_{star} for the FMMC problem on a star grah are given by

$$\lambda = \left(1, \underbrace{\frac{n-2}{n-1}, \ldots, \frac{n-2}{n-1}}_{n-2}, -\frac{1}{n-1} \right)$$

for $n > 2$. For figure 8 we have defined

$$P_1 = \begin{pmatrix} 0.1 & 0.2 & 0.2 & 0.2 & 0.3 \\ 0.2 & 0.8 & 0 & 0 & 0 \\ 0.2 & 0 & 0.8 & 0 & 0 \\ 0.2 & 0 & 0 & 0.8 & 0 \\ 0.3 & 0 & 0 & 0 & 0.7 \end{pmatrix} \tag{53}$$

and

$$P_1 = \begin{pmatrix} 0.5 & 0.1 & 0.1 & 0.1 & 0.2 \\ 0.1 & 0.9 & 0 & 0 & 0 \\ 0.1 & 0 & 0.9 & 0 & 0 \\ 0.1 & 0 & 0 & 0.9 & 0 \\ 0.2 & 0 & 0 & 0 & 0.8 \end{pmatrix} \tag{54}$$

and P_{opt} is the optimal transition probability matrix on a star graph.

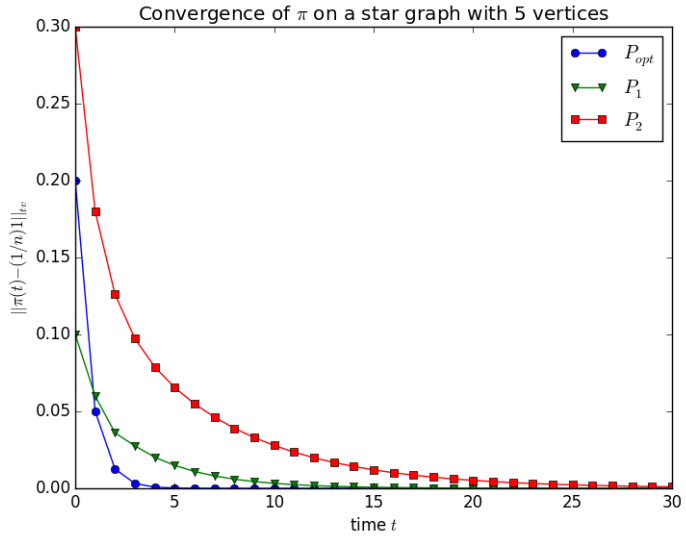

FIGURE 8. The plot shows the convergence of three Markov chains on a star graph with the transition probability matrices P_{opt}, P_1 and P_2. The star graph has 5 vertices and the initial probability distribution $\pi(0)$ is choosen to be $\pi(0) = (1, 0, 0, 0, 0)$. We use the total variation distance to measure the distance between the two distributions $\pi(t)$ and $(1/n)\mathbf{1}$ at a time t.

2. Subgradient method

In this section we are going to apply the subgradient method compare the performance on different types of graph. First we will take a closer look at different step-size rules when we run the algorithm. Next, we will measure the time when we solve the subgradient method on randomly generated graphs.

2.1. Step-size rules. We introduced the subgradient method in section 6. When we use the algorithm to find the best possible solution, we have the choice of choosing different step-size rules. The rules we looked at in the background theory of the method, were the constant step-size, constant step length, square summable but not summable and the nonsummable diminishing step rule. The choice of step-size plays an important role when we want fast convergence of the algorithm.

Now we will solve the FMMC with the subgradient method by testing the four step-size rules (see [7]):

(1) constant step-size, $\alpha_k = h$
(2) constant step length, $\alpha_k = h/\|g^{(k)}\|_2$
(3) square summable but not summable, $\alpha_k = a/(b+k)$,where $a > 0$ and $b \geq 0$
(4) nonsummable dimminishing, $\alpha_k = a/\sqrt{k}$, where $a > 0$

To view the progress of the subgradient method for different step-size rules we will solve the FMMC problem on a cycle graph with 5 vertices. The results of the step-size rules given above are shown in figures 9, 10, 11 and 12, respectively, where the x-axis shows the iteration number of the subgradient method and y-axis the gap between the optimal solution and the solution found for the k-th iteration.

In figure 9, we have choosen three values of h, 0.02, 0.01 and 0.005 for the constant step-size rule $\alpha_k = h$ and did 100 iterations in order to obtain best solution. We see that the choice of h value determines how fast the algorithm converges to the optimal value. Especially, the gap between the k-th iterate $f^{(k)}$ and the optimal decreases to zero linearly. $h = 0.02$ converges fastest out of the three choices of h that we provided.

The same behavior as in figure 9 can be recognized in figure 10. We see that the gap decreases linearly to zero and then all the solution oscillates around the optimal value. Here, we can also see that the choices of h gives different convergence. For $h = 0.02$ the gap tends to zero after approximately 25 iterations, $h = 0.01$ after 50 iterations and $h = 0.005$ after over 100 iterations.

3. Random generated graphs

We will show a method for generating random graphs using the procedure described in Boyd, Diaconis, and Xiao [2].

To generate the graphs, we use the description in [2] (see section 3.3 in [2]) by the following:

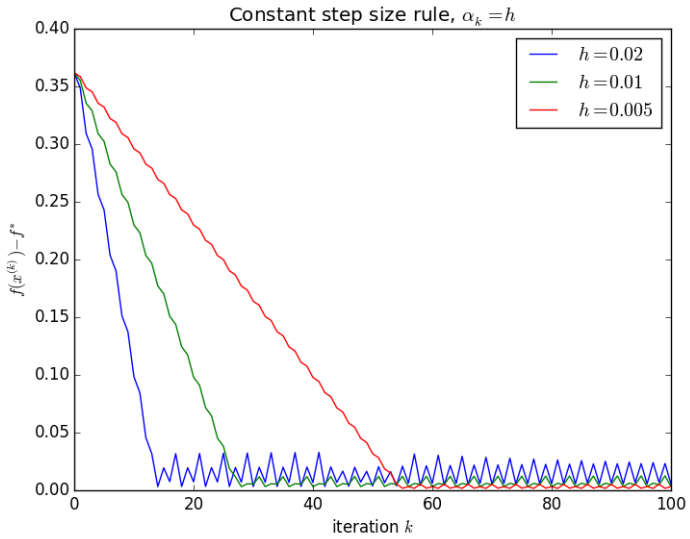

FIGURE 9. The figure shows the progress of the gap between the k-th iteration $f^{(k)}$ and the optimal solution f^* for 100 iterations when solving the FMMC problem on a cycle graph with 5 vertices. The iteration number is along the x-axis and the gap is on the y-axis. $\alpha_k = h$ for $h = 0.02$ (blue), $h = 0.01$ (green) and $h = 0.005$ (red).

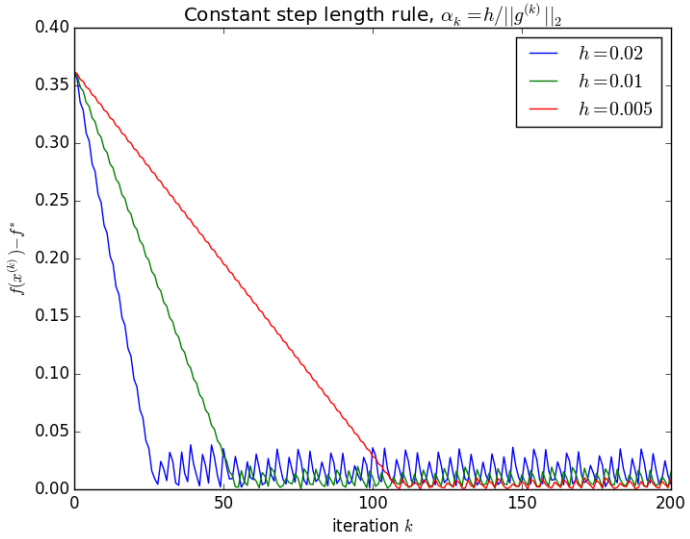

FIGURE 10. The figure shows the progress of the subgradient method on a cycle graph with 5 vertices using the step length size rule $\alpha_k = h/\|g^{(k)}\|_2$ when $h = 0.02, 0.01, 0.005$.

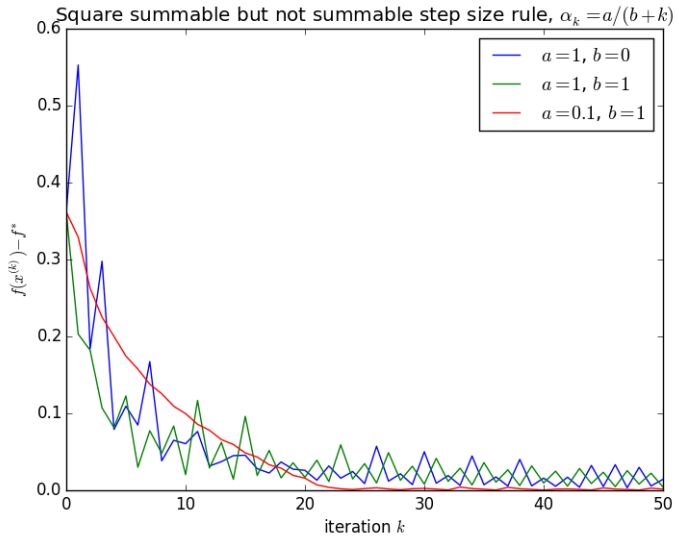

FIGURE 11. The figure shows the progress of the subgradient method on a cycle graph with 5 vertices using the square summable but not summable step-size rule, $\alpha_k = a/(b+k)$ for $a = 1, 1, 0.1$ and $b = 0, 1, 1$.

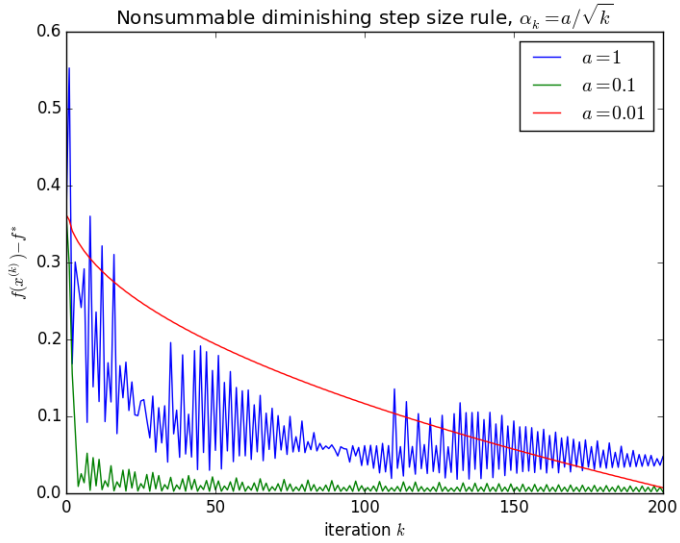

FIGURE 12. The figure shows the progress of the subgradient method on a cycle graph with 5 vertices using the nonsummable diminishing step-size rule $\alpha_k = a/\sqrt{k}$ for $a = 1, 0.1, 0.01$.

(1) Generate a random symmetric matrix $R \in \mathbb{R}^{n\times n}$, where n is the number of vertices and R_{ij} are independent and uniformly distributed on the interval $[0, 1]$ for $i \leq j$.

(2) Construct graph. Choose threshold value $c \in [0, 1]$ and connect two vertices i and j with an edge when $i \neq j$ if $R_{ij} \leq c$. Self-loop are also added to the graph.

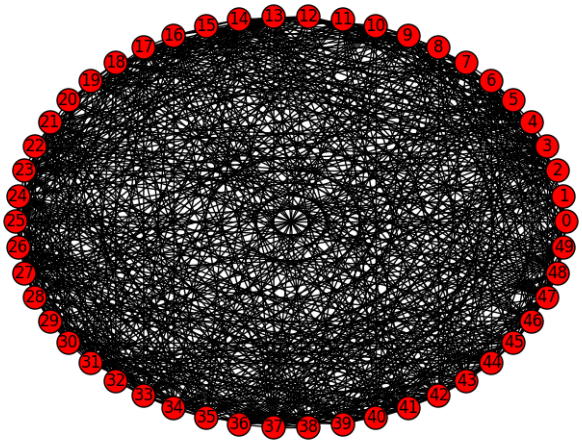

FIGURE 13. The figure shows a randomly generated graph with 50 vertices and 686 edges.

In this section we will consider the eigenvalue distribution of the transition probability matrix P. Since the eigenvalue of the matrix plays such an important role for fast convergence, we would like to find the distribution given the graph.

By following the rules above we will obtain a monotone family of graphs if we increase the value of c from 0 to 1. Large values of c contain all the edges of smaller values of c. The graphs in the FMMC problem must be connected, so we have to be careful when we choose the smallest value of c to get connected graphs. The Python code for generating the random graphs is given in A.6.

In figure 14, we have plotted the eigenvalues with respect to the iteration number k when we solve the FMMC problem on a randomly generated, connected graph with 10 vertices and 40 edges using the subgradient method.

Figure 15 shows the comparison of the SLEM value for the optimal transition probability matrix, and the matrices generated by the maximum-degree algorithm and the Metropolis-Hastings algorithm for various number of edges.

Figure 16 shows the distribution of the eigenvalue for a given value of c. We see that each of the plots has 1 as eigenvalue and that the matrices have some negative eigenvalues. By comparison, the Metropolis-Hastings chain has smaller second eigenvalue than the maximum-degree chain, when we consider the ordered eigenvalues, but it is not smaller than the second eigenvalue of the fastest mixing chain.

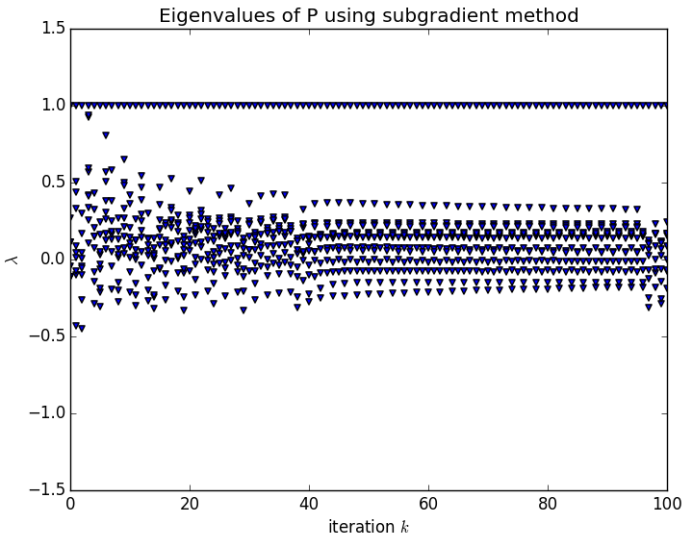

FIGURE 14. The scatter plot shows the distribution of the eigenvalues on a connected graph with respect to the iteration number using the subgradient method.

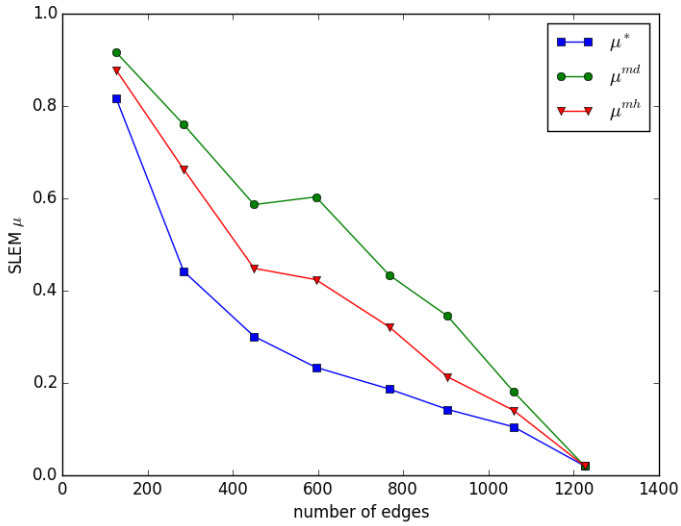

FIGURE 15. The plot shows the SLEM value μ for the optimal transition probability matrix, the matrix generated using the maximum-degree algorithm and the Metropolis-Hasting algorithm for random graphs with respect to the number of edges.

In table 7, shows the run time of solving randomly generated graphs with 100 vertices. We used the subgradient method, where we ran 200 iterations,

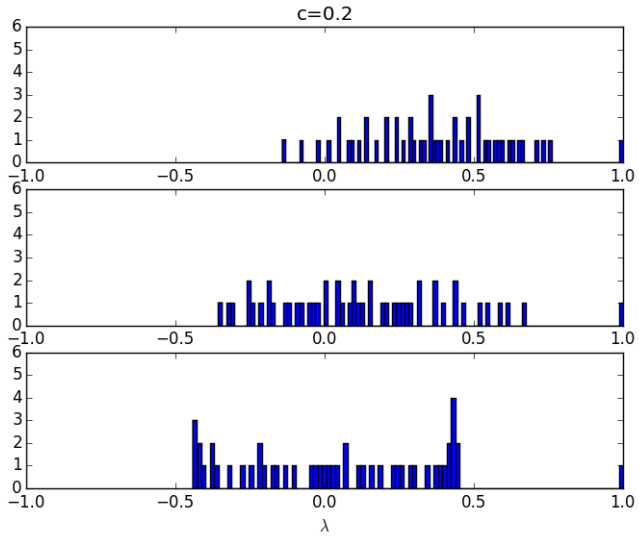

FIGURE 16. The figure shows the eigenvalue distristribution of the fastest mixing for randomly generated graph with 50 vertices. A

Time (in seconds)	Number of edges
70.5	2681
69.3	2619
69.9	2632
68.3	2574
67.1	2528

TABLE 7. The table shows the run time of solving randomly generated graphs with 100 vertices using the subgradient method.

initialized a starting point with the maximum-degree chain, and used the $\alpha_k = 1/\sqrt{k}$ as stepsize rule.

CHAPTER 5

Comparison of convex optimization solvers

We will in this chapter do some test run FMMC problem for small graphs using the primal-dual interior-point method from [9], and compare it to the CVXOPT package for convex optimization for the Python language. The Python implementation of the primal-dual interior-point method (XZ-method and $XZ + ZX$-method) can be found in the appendix.

Let us recall the FMMC problem which will be useful for the upcoming two sections. The problem

$$\begin{array}{ll} \text{minimize} & s \\ \text{subject to} & -sI \preceq P - (1/n)\mathbf{1}\mathbf{1}^T \preceq sI \\ & P \geq 0,\ P\mathbf{1} = \mathbf{1},\ P = P^T, \\ & P_{ij} = 0,\ (i,j) \notin \mathcal{E}. \end{array} \tag{55}$$

where the variables are matrix P and scalar s.

1. Primal-dual interior-point methods

In chapter 2, section 5.2 we introduced a primal-dual interior-point methods for solving semidefinite programs. We will model the FMMC problem by constructing the matrices and scalars such that we can solve the problem with XZ- and $XZ + ZX$-method. Then we will give a time complexity analysis of the implementation for modeling the problem.

1.1. Modeling the FMMC problem. In this section we have made an effort of going into detail about how we make the formulation of the FMMC problem as a semidefinite program stated in equation 39. To be able to solve the problem using the primal-dual interior-point methods the problem has to be on a specific form which we will explain.

In order to use the primal-dual interior-point methods, XZ- and $XZ + ZX$-method, we will have to formulate the problem on the form

$$\begin{array}{ll} \text{minimize} & \langle C, X \rangle \\ \text{subject to} & \langle A_i, X \rangle = b_i,\ i = 1, \ldots, m, \\ & X \succeq 0 \end{array} \tag{56}$$

where matrix X is the variable.

We will now go through the construction of the matrices A_i, scalars b_i and matrix C for modeling the FMMC problem on the form in equation 56 for $i = 1, \ldots, m$.

First, we will define some notation to make it easier to see the construction of the parameter matrices A_i. Define $\text{diag}(\cdot)$ and $\text{vec}(\cdot)$. We let

$\text{diag}(A_1, \ldots, A_n)$ for matrices $A_1, \ldots, A_n$ denote the block diagonal matrix on the form

$$\text{diag}(A_1, \ldots, A_n) := \begin{pmatrix} A_1 & & \\ & \ddots & \\ & & A_n \end{pmatrix}$$

where the non-block-diagonal are zero. The size of the matrix $\text{diag}(A_1, \ldots, A_n)$ is determined by the input arguments which is given by $\sum_{i=1}^{n} m_i \times \sum_{i=1}^{n} n_i$ where m_i and n_i are the dimension of the matrix $A_i \in \mathbb{R}^{m_i \times n_i}$.

Let $\text{vec}(X)$ denote the vectorization of the matrix X, which means that the columns of X are stacked to form a vector in $\mathbb{R}^{n^2}$. Let $X = [x_1, \ldots, x_n]$ where x_i is the column of the matrix then

$$\text{vec}(X) := \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

Let n be the number of vertices in the graph $\mathcal{G}$. Let $\mathbf{0} \in \mathbb{R}^{n \times n}$ denote a zero matrix where all the entries are zero.

In order to model the FMMC problem on the form as in equation 56, we will consider constraints in equation 55.

The variable in equation 56 is X which is symmetric and satisfies $X \succeq 0$,i.e, X postive semidefinite. Let $X \in \mathbb{R}^{(n+1)^2 \times (n+1)^2}$ and set

$$X = \text{diag}(sI - P + (1/n)\mathbf{1}\mathbf{1}^T, sI + P - (1/n)\mathbf{1}\mathbf{1}^T, \text{diag}(\text{vec}(P)), s)$$

where $\text{vec}(P)$ is the vectorization of P. $\text{diag}(\cdot)$ means that we put the two matrices $sI - P + (1/n)\mathbf{1}\mathbf{1}^T$ and $sI + P - (1/n)\mathbf{1}\mathbf{1}^T$ as block diagonals on X, the vector $\text{vec}(P)$ and the scalar t on the remaining diagonal on X. Let X_1 and X_2 be the two block diagonal matrices of X. Provided that the equalities $X_1 = sI - P + (1/n)\mathbf{1}\mathbf{1}^T$ and $X_2 = sI + P - (1/n)\mathbf{1}\mathbf{1}^T$ hold, we have that $X \succeq 0$. Since X is positive semidefinite implies that the following hold:

$$\begin{aligned} sI - P + (1/n)\mathbf{1}\mathbf{1}^T &\succeq 0 \\ sI + P - (1/n)\mathbf{1}\mathbf{1}^T &\succeq 0 \\ P &\geq 0 \\ s &\geq 0 \end{aligned}$$

We will now focus on the contraints of the problem by determine the parameter matrices A_i and the scalar b_i for $i = 1, \ldots, m$ in order to set the equality constraints $\langle A_i, X \rangle = b_i$ for $i = 1, \ldots, m$. The total number of equality contraints is

$$m = 2n^2 + n + n(n-1)/2 + k$$

where k is the number of edges not in $\mathcal{G}$. The $2n^2$ equality contraints come from the equalities of the block diagonal matrices X_1 and X_2. Furthermore, it requires n equations to set the contraints for $P\mathbf{1} = \mathbf{1}$, $n(n-1)/2$ equations for the symmetry and k contstraints for the edges which is not in $\mathcal{G}$. For each of the equality constraints we will define a matrix A_i and a scalar b_i.

We will construct the $A_i \in \mathbb{R}^{(n+1)^2 \times (n+1)^2}$ matrix and the corresponding scalar b_i based on the equality constraints of the FMMC problem. First, we will take care of the equality for the block matrices X_1 and X_2, and then continue with the row sum (or column sum) of P which must be equal to 1. Next, the symmetry of P has to be hold and finally the probability of edges not in the graph is set to 0.

We get started with the block matrix X_1 which satisfies $X_1 = sI - P + (1/n)\mathbf{1}\mathbf{1}^T$. Elementwise we have that if $i = j$ then $(X_1)_{ii} + P_{ii} - s = 1/n$, otherwise $(X_1)_{ij} + P_{ij} = 1/n$. The matrix we are about to construct for these equalities consists of -1, 0 or 1 at the entry. To simplify the notation, we will make use of the diag$(\cdot)$ and define a matrix $E \in \mathbb{R}^{n \times n}$. Say we want to find the equality constraint for the pair (i, j) of X_1 then let $E_{ij} = 1$ and all other entries of E be zero. The parameter matrix A_k that models the equality of $(X_1)_{i,j}$ is given by

$$A_k = \begin{cases} \text{diag}(E, \mathbf{0}, \text{diag}(\text{vec}(E)), -1) & i = j \\ \text{diag}(E, \mathbf{0}, \text{diag}(\text{vec}(E)), 0) & \text{otherwise} \end{cases} \tag{57}$$

The corresponding scalar $b_k = 1/n$. Similar construction can be done for the equalities for the block matrix X_1. Let E be as above, then the parameter matrix A_k becomes

$$A_k = \begin{cases} \text{diag}(\mathbf{0}, E, \text{diag}(\text{vec}(-E)), 1) & i = j \\ \text{diag}(\mathbf{0}, E, \text{diag}(\text{vec}(-E)), 0) & \text{otherwise} \end{cases} \tag{58}$$

with corresponding scalar $b_k = -1/n$.

Now we will construct the parameter matrix A_k for the equality $P\mathbf{1} = \mathbf{1}$. To make sure that the row sum is 1 for the row r in P, we let the row of E be 1 and the rest of the entries are zero. Again, we will use the diag$(\cdot)$ to simplify the construction. So, let the parameter constraint for the row sum be given by

$$A_k = \text{diag}(\mathbf{0}, \mathbf{0}, \text{diag}(\text{vec}(E)), 0) \text{ and } b_k = 1 \tag{59}$$

To ensure symmetry of the P, $n(n-1)/2$ equality contraints have to hold. The equalities can be described as $P_{ij} = P_{ji}$ for $(i, j) \in \mathcal{E}$, $i \neq j$, thus we can represent the equalities by setting $\langle A_k, X \rangle = P_{ij} - P_{ji} = 0$. We can neglect the equation when $i = j$ because it becomes trivial, therefore we will only handle the equality for entries on the non-diagonal. Let us derive the symmetry equality constraint for the entry (i, j) when $i \neq j$. Let $E \in \mathbb{R}^{n \times n}$ be defined by $E_{ij} = 1$ and $E_{ji} = -1$ and zero otherwise. The parameter matrix A_k can be constructed by

$$A_k = \text{diag}(\mathbf{0}, \mathbf{0}, \text{diag}(\text{vec}(E)), 0) \tag{60}$$

with the corresponding scalar $b_k = 0$.

Lastly, we will constuct the equality constraint for the probability for edges not contained in the graph $\mathcal{G}$. Say we have a pair (i, j) that is not in graph i.e $(i, j) \notin \mathcal{E}$. Let $E \in \mathbb{R}^{n \times n}$ be given as $E_{ij} = 1$ such that the parameter constraint is

$$A_k = \text{diag}(\mathbf{0}, \mathbf{0}, \text{diag}(\text{vec}(E)), 0) \text{ with } b_k = 0 \tag{61}$$

We set $\langle A_k, X\rangle = P_{ij} = 0$ for a pair (i, j). The number of equality contraints is determined by the graph. For sparse graphs we will have more equality constraints than for dense graphs.

By following the steps above we will be able to construct the constraints for the FMMC problem. We can see that the matrices A_k will be sparse because it contains a lot of zeros. The final step is to determine the objective function such that $\langle C, X\rangle = s$. So let $C \in \mathbb{R}^{(n+1)^2\times(n+1)^2}$ and set the last diagonal element of C to be 1, i.e,

$$C = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} \tag{62}$$

Now that we have given the details of the parameter matrices of the problem, we will provide an overview of the procedure in algorithm 1. First we make the parameter matrices A_k and the scalars b_k for the equality constraints, then we construct C for the objective function. We initialize the dual variables, the vector y and the matrix Z, for starting point of the primal-dual interior-point methods. Once the matrices and vectors are constructed, the FMMC problem is modeled, and we can solve the problem using the XZ-method or $XZ + ZX$-method. A Python implementation of the FMMC problem using the methods can be found in A.3 and A.4.

Algorithm 1: Modeling the FMMC problem and solve using primal-dual interior-point solver

Data: Undirected, connected graph
Result: Transition probability matrix P and its SLEM value s
1 Make A_k and b_k for $k = 1, \ldots, m$ as in (57)-(61)
2 Make C as in (62)
3 Initialize dual variables y (vector) and Z (matrix)
4 Call *XZ-method* or *$XZ + ZX$-method* with C, X, y, Z, A_k and b_k

1.2. Time complexity analysis of the XZ-method and the $XZ+ZX$-method. In this section, we are going to compare the performance of the two algorithms, XZ-method and $XZ + ZX$-method. We will look into the parameter τ which determines the steplength for each iteration, and see how different choices of τ affect the steplength, and therefore the number of iterations to solve the problem. Finally, we will give a complexity analysis based on our implementations found in the appendix A.

We will now look at the time complexity of the implementation for modeling the FMMC problem. In our implementation, the diagonalization of the matrices (the function *diag*) has complexity of $\mathcal{O}(n^4)$. To model the contraint $-sI \succeq P - (1/n)\mathbf{1}\mathbf{1}^T \succeq sI$, the complexity is $\mathcal{O}(n^6)$.

If we consider the solver the numpy function *numpy.linalg.solve*, it solves a linear system in $\mathcal{O}(m^3)$ time. In the implementation of finding the search direction (see function *XZ_search*), it takes as input a matrix $X \in \mathbb{R}^{(n+1)^2\times(n+1)^2}$, maps it to $\mathbb{R}^{(n+1)^4}$. So the overall complexity for finding a search direction is $\mathcal{O}(m^3)$ where $m = n^4$. By looking at the complexity of just finding a

search direction we may conclude that the implementions are slow.

A simple steplength rule is given in Alizadeh, Haeberly, and Overton [9] for the primal-dual interior-point methods XZ-method and $XZ + ZX$-method, which is choosing parameter τ, $0 < \tau < 1$ and defining $\alpha = \min(1, \tau\hat{\alpha})$ where $\hat{\alpha} = \sup\{\bar{\alpha} : X + \bar{\alpha}\Delta X \succeq 0\}$, and $\beta = \min(1, \tau\hat{\beta})$ where $\hat{\beta} = \sup\{\bar{\beta} : X + \bar{\beta}\Delta X \succeq 0\}$. We let α be the steplength of the primal variable X and β for the dual variables y and Z, such that the update of the iterates become $X \to X + \alpha\Delta X$, $y \to y + \beta\Delta y$ and $Z \to Z + \beta\Delta Z$.

We will now report the result of the some test runs of different choices of σ and τ on a star graph. Then we will measure the time usage of our implementation of the primal-dual interior-point method on randomly generated graphs and on the four small graphs in 4. All the test runs reported in the tables are initialized with $(X_0, y_0, Z_0) = (I, 0, I)$ as starting point.

Iterations	σ	τ
10	0.001	0.9
10	0.01	0.9
11	0.05	0.9
11	0.1	0.9
12	0.15	0.9
13	0.2	0.9

TABLE 1. The table shows the number of iterations to solve the FMMC problem on a star graph with $n = 5$ vertices using the XZ-method for $\tau = 0.9$.

Iterations	τ	σ
31	0.5	0.1
24	0.6	0.1
19	0.7	0.1
14	0.8	0.1
11	0.9	0.1
9	0.99	0.1

TABLE 2. The table shows the number of iterations used to solve the FMMC problem on a star graph with $n = 5$ vertices using the XZ-method for $\sigma = 0.1$.

Table 1 and table 2 show the number of iterations required by XZ-method of finding the transition probability matrix P that gives the fastest mixing. We have tried different values of τ and $\sigma = 0.25$. And we see that the number of iterations are minimized when τ is close to 1 and σ is close to 1.

In table 3, we see the measured time to find the fastest mixing Markov chain on a randomly generated graph with $n = 5$ vertices. When the graph becomes more dense, it takes less time to solve than for sparse graphs.

Number of edges	Time (in seconds)	Iterations
5	22.7	18
7	26.4	21
8	24.5	19
10	19.4	15

TABLE 3. The table shows the time usage and number of iterations to solve the FMMC problem on randomly generated graphs with $n = 5$ vertices using the XZ-method.

Graph	Number of edges	Time (in seconds)	Iterations
(A)	3	6.3	21
(B)	4	4.6	15
(C)	6	17.6	14
(D)	8	17.7	14

TABLE 4. The table shows the time usage and number of iterations to solve the FMMC problem on the four small graphs that we looked on, in chapter 4, using the XZ-method.

Graph	Number of edges	Time (in seconds)	Iterations
(A)	3	4.3	14
(B)	4	4.4	14
(C)	6	18.2	14
(D)	8	18.2	14

TABLE 5. The table shows the time usage and number of iterations to solve the FMMC problem on the four small graphs defined earlier, in chapter 4, using the $XZ + ZX$-method.

The result of table 4 and table 5 shows that the measured time for the four small graphs in chapter 4. For the test runs we have set the tolerance $\text{tol} = 10^{-6}$, $\sigma = 0.2$ and $\tau = 0.8$.

2. Convex optimization solver CVXOPT

Similar to the previous section, we will start by modeling the FMMC problem on the following as the semidefinite program. The details of the parameters will be stated, and then a overview of the steps will be given in a pseudocode. An analysis of the complexity of modeling the FMMC will be stated by looking into our implementation (see appendix A).

The goal is to formulate the FMMC as the semidefinite program on the form:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & G_0 x + s_0 = h_0 \\ & G_k x + \text{vec}(s_k) = \text{vec}(h_k),\ k = 1, \ldots, N \\ & Ax = b \\ & s_k \succeq 0,\ k = 0, \ldots, N \end{array} \tag{63}$$

For the FMMC problem formulated as a semidefinite program the variables are s and P, where s is the SLEM value and P the transition probability matrix. In order to formulate the SDP formulation of the problem we would let $x \in \mathbb{R}^{n^2+1}$ denote the variables of the FMMC, set $x = (p_1, \ldots, p_n, s)$, where p_i denotes the i-th column of the matrix P and s is the spectral norm of $P - (1/n)\mathbf{1}\mathbf{1}^T$. To formulate the constraints $P\mathbf{1} = \mathbf{1}$, $P = P^T$ and $P_{ij} = 0$ if $(i,j) \notin \mathcal{E}$, we will let $A \in \mathbb{R}^{m \times (n^2+1)}$ and $b \in \mathbb{R}^m$ where $m = n + n(n-1)/2 + k$, and k is the number of non-edges of the graph, such that the linear system $Ax = b$ models the equality constraints of P mentioned above. We will simplify the construction of the matrix A by dividing the matrix into three block matrices, where each block matrix represents the equality constraints of P. Construct A by

$$A = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \end{pmatrix}$$

where $A_1 \in \mathbb{R}^{n \times (n^2+1)}$, $A_2 \in \mathbb{R}^{n(n-1)/2 \times (n^2+1)}$ and $A_3 \in \mathbb{R}^{n\varepsilon \times (n^2+1)}$ are block diagonals. A_1 models the constraint $P\mathbf{1} = \mathbf{1}$, A_2 models the constraint of symmetry of P and A_3 the constraint for the transition probability on non-edges. We will now define the block matrices in the given order as above so we get started with A_1. Let

$$A_1 = \begin{pmatrix} \overbrace{1 \cdots 1}^{n} & & \\ & \ddots & \\ & & \overbrace{1 \cdots 1}^{n} \end{pmatrix} \tag{64}$$

where only the non-zero elements are given.

For symmtry of P, let each row k of A_2 represent the equality $P_{ij} = P_{ji}$ such that $A_2 x = P_{ij} - P_{ji} = 0$ for an edge (i,j) for $0 \leq i < n, 0 \leq j < i$, we can set the non-zero entries of A_2 to be

$$(A_2)_{k,in+j} = 1 \text{ and } (A_2)_{k,i+jn} = -1 \tag{65}$$

There is $n(n-1)/2$ equalities for representing the symmetry of P.

Let $n_{\mathcal{E}}$ denote the number of edges not in the graph. To represent the constraint where the transition probability of a non-edge is set to zero, we use A_3. For the row k of A_3 and an edge $(i,j) \notin \mathcal{E}$ we set the non-zero entry of A_3 to be

$$(A_3)_{k,i+jn} = 1 \tag{66}$$

When we put together the block matrices A_1, A_2 and A_3, the resulting matrix A has entries $-1, 0$ or 1 and the corresponding vector $b \in \mathbb{R}^m$ matrix can be defined as

$$b = (\overbrace{1, \ldots, 1}^{n}, \overbrace{0, \ldots, 0}^{n(n-1)/2+k}) \tag{67}$$

To modify $P \geq 0$ we will modify the equation $G_0x + s_0 = h_0$ provided that s_0 is positive. We want that $x \succeq 0$ which means that $x = s_0$. If we choose $h_0 \in \mathbb{R}^{n^2+1}$ such that $h_0 = 0$ (zero vector), we obtain that $G_0 = -I$ as $G_0x + s_0 = G_0x + Ix = (G_0 + I)x = 0 \Leftrightarrow G_0 = -I$. Let $G_0 \in \mathbb{R}^{(n^2+1)\times(n^2+1)}$ and set

$$G_0 = -I \text{ and } h_0 = 0 \tag{68}$$

where I is the identity matrix and 0 is the zero vector.

For the FMMC problem on the form in equation 63, let $N = 2$. Recall the constraints $P - sI \preceq (1/n)\mathbf{1}\mathbf{1}^T$ and $-P - sI \preceq -(1/n)\mathbf{1}\mathbf{1}^T$ of the problem. To make the formulation easier we will define a slack variable $Z_1, Z_2 \in \mathbb{R}^{n\times n}$ to get $P - sI + Z_1 = (1/n)\mathbf{1}\mathbf{1}^T$ and $-P - sI + Z_2 = -(1/n)\mathbf{1}\mathbf{1}^T$ under the condition that $Z_1, Z_2 \succeq 0$. Let $G_1, G_2 \in \mathbb{R}^{n^2\times(n^2+1)}$. We will now indicate the the non-zero elements of G_1 and G_2. We start with G_1. For i, j $(0 \leq i, j < n)$ we set

$$(G_1)_{i+jn,i+jn} = 1 \tag{69}$$

and for the last column of G_1 if $i = j$ we set

$$(G_1)_{i+jn,n^2+1} = -1 \tag{70}$$

Almost the same procedure can be done for G_2, we set

$$(G_2)_{i+jn,i+jn} = -1 \tag{71}$$

for $0 \leq i, j < n$ and for $i = j$ set

$$(G_2)_{i+jn,n^2+1} = -1 \tag{72}$$

Let $h_1, h_2, s_1, s_2 \in \mathbb{R}^{n\times n}$ and set

$$h_1 = (1/n)\mathbf{1}\mathbf{1}^T \text{ and } h_2 = -(1/n)\mathbf{1}\mathbf{1}^T \tag{73}$$

and let $s_1 = Z_1$, $s_2 = Z_2$. By constructing G_1,G_2,h_1 and h_2 on the form as above the equation $G_1x + \text{vec}(s_1) = \text{vec}(h_1)$ is equivalent to $P - sI + Z_1 = (1/n)\mathbf{1}\mathbf{1}^T$ and similar for $G_2x + \text{vec}(s_2) = \text{vec}(h_2)$ is equivalent to $-P - sI + Z_2 = -(1/n)\mathbf{1}\mathbf{1}^T$.

Finally, let $c \in \mathbb{R}^{n^2+1}$ and set the non-zero entry of the vector to be 1 at its last entry, that is,

$$c_i = \begin{cases} 1 & i = n^2 + 1 \\ 0 & \text{otherwise} \end{cases}$$

We have made a summary in form of a pseudocode to make it easier to get an overview of the procedure using the CVXOPT package to solve the FMMC problem. The pseudocode can be found in Algorithm 2. It takes use through the construction of the matrices and vectors required for modeling the FMMC problem before we call the solver. First we make the matrix

A and vector b for the m equality constraints, which takes care of the row sum sums to one, symmetry of the transition probability matrix and the probability on the non-edges of the graph. Next, we make G_1, G_2 and h_1, h_2 to model the contraint $-sI \preceq P - (1/n)\mathbf{1}\mathbf{1}^T \preceq sI$. After this we pack the matrices G_1, G_2 and h_1, h_2 into a list G and list h. When all the matrices and vectors are made, we call the *cvxopt.sdp* to solve our problem.

Algorithm 2: Modeling the FMMC problem and solve it using a CVXOPT solver

Data: undirected, connected graph
Result: Optimal transition probability matrix P and its SLEM value μ for the FMMC problem on graph

1 Make matrix A and b as in (64), (65), (66) and (67)
2 Make matrix G_1, G_2 as in (69), (70) (71) and (72)
3 Make matrix h_1, h_2 as in (73)
4 Make vector c
5 List $G = [G_1, G_2]$ and $h = [h_1, h_2]$
6 Set G_0 and h_0 as in (68)
7 Call cvxopt.sdp with c, G, h, G_0, h_0, A and b

2.1. Time complexity analysis. In this section, we will give an analysis of the time complexity of the implementation of solving the FMMC problem with the convex optimization solver CVXOPT. The implementation can be found in appendix A.

First, we will take a closer look at the complexity to construct the constraints of equation 55. For the constraint $P\mathbf{1} = \mathbf{1}$ it requires $\mathcal{O}(n)$ complexity. To model the symmetry of P we have $\mathcal{O}(n^2)$ and for edges not in the graph the complexity is $\mathcal{O}(n^2)$. For creating the positive semidefinite constraints we we need $\mathcal{O}(n^2)$, and $\mathcal{O}(1)$ to construct the objective function c. If we look at the total complexity of modeling the constaints, inclusive the objective function, we have

$$n + n^2 + n^2 + n^2 + 1 = 3n^2 + n + 1$$

so the complexity is $\mathcal{O}(n^2)$. The worst case time complexity of the initialization of the matrices and vectors is $\mathcal{O}(n^4)$.

Number of edges	Time (in seconds)
52	23.4
178	23.0
304	19.6
435	8.6

TABLE 6. The table shows the time usage and number of iterations to solve the FMMC problem on randomly generated graphs with $n = 30$ vertices using CVXOPT.

Table 6 shows that the time for solving the FMMC problem reduces when the graphs have many edges. The number of edges not in the graph

determines the size of the problem set. Because of this, it may be appropriate to say that the problem can be solved faster for dense graphs than sparse graphs.

Number of edges	Time (in seconds)	μ
3	0.060000	0.666667
4	0.000000	0.750000
5	0.010000	0.800000
6	0.010000	0.833333
4	0.010000	0.335236
5	0.010000	0.465477
6	0.010000	0.605225
7	0.010000	0.677831
9	0.040000	0.789770
10	0.070000	0.827593
11	0.380000	0.853186
12	0.410000	0.875973

TABLE 7. The table shows the SLEM values for the FMMC problem on cycle graphs using the CVXOPT.

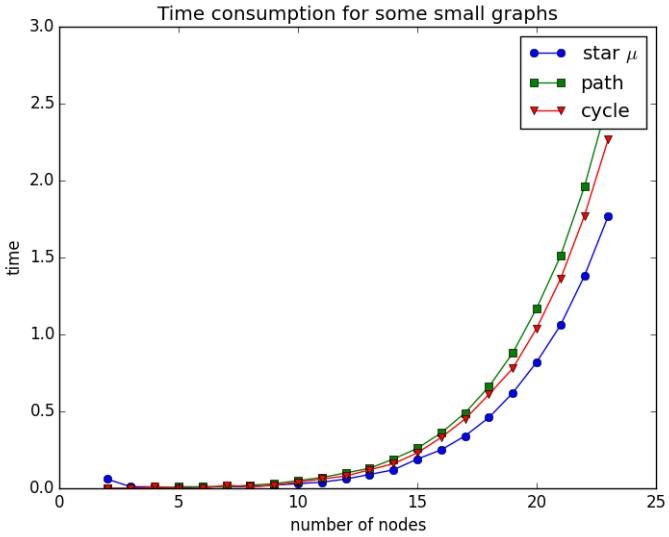

FIGURE 1. The figure shows the run time for the FMMC problem using CVXOPT. The graphs that we solved for are paths, cycles and stars.

We would like to show When we look at our implementation of the primal-dual interior-point method againtst the semidefinite programming package CVXOPT to solve SDP, there is no doubt that the latter solver performs better. Both implementations require that we model the problem in different ways, which is described in the previous chapter. In the implementation we made of the XZ-method and $XZ + ZX$-method, we have

used a numerical solver in Python to solve a linear system to solve the Newton step, which has a complexity $\mathcal{O}(n^3)$. The total complexity to find a search direction is $\mathcal{O}(n^6)$, as we discussed previously. Another disadvantage is that the matrices of the problem becomes very sparse, especially when the number of vertices becomes large.

To round off this chapter we will try to answer the question about card shuffling and cup shuffling with some example from chapter 3, section 2; how long do we have to shuffle to get randomness? To answer this, we will look at the card shuffle of a deck of 3 cards as in example 2.1. Let the graph be as defined in the example, then we find that the optimal transition probability matrix P^* is given by

$$P^* = \begin{pmatrix} 0.222 & 0 & 0.333 & 0.222 & 0.222 & 0 \\ 0 & 0.222 & 0.222 & 0 & 0.333 & 0.222 \\ 0.333 & 0.222 & 0.222 & 0 & 0 & 0.222 \\ 0.222 & 0 & 0 & 0.222 & 0.222 & 0.333 \\ 0.222 & 0.333 & 0 & 0.222 & 0.222 & 0 \\ 0 & 0.222 & 0.222 & 0.333 & 0 & 0.222 \end{pmatrix}$$

We use the implementation that we introduced in the beginning of this section that simulates a Markov chain for a period of discrete time, given an initial probability distribution $\pi(0)$ and a transition probability matrix P. The result shows that the probability distribution reaches the uniform equilibrium distribution $(1/n)\mathbf{1}$ after $t = 15$ time steps by computing $\pi(t + 1) = \pi(t)P$ for $t \geq 0$.

EXAMPLE 2.1 (Two rules of cup shuffle with 5 cups). *For the cup shuffle of 5 cups. Let us consider two rules for shuffling the cups represented in the the graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ (see figure 2 and figure 3). A question we may ask is, which rule is the best way to shuffle?*

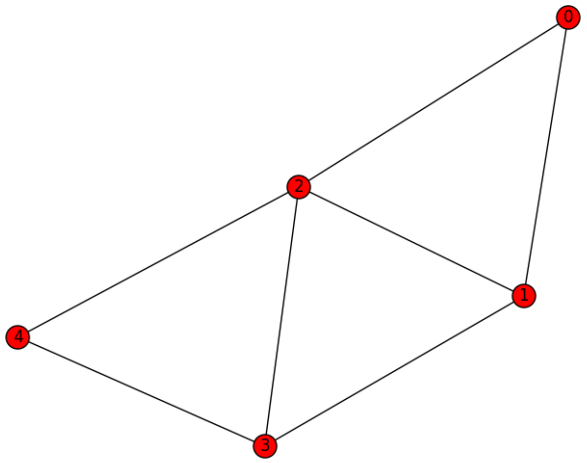

FIGURE 2. Graph $\mathcal{G}_1$

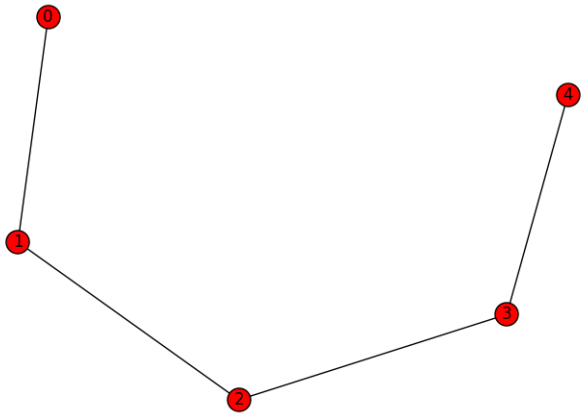

FIGURE 3. Graph $\mathcal{G}_2$

We will use the CVXOPT to solve the FMMC problems for the two graphs, an then we simulate for the probability distributions to compare the convergence towards $(1/n)\mathbf{1}$ *for* $n = 5$.

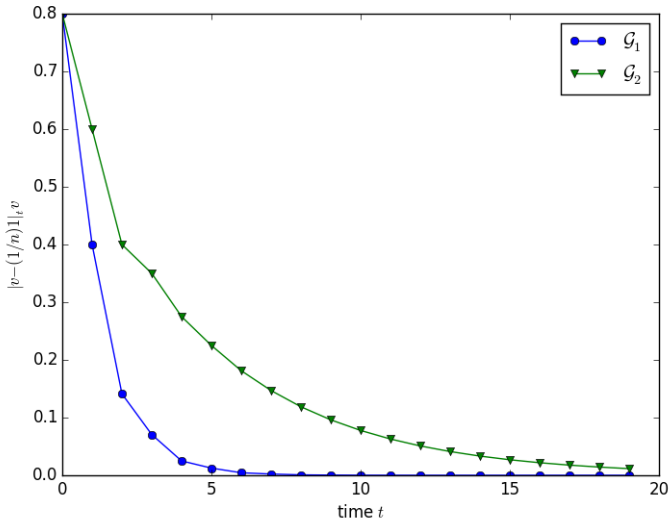

FIGURE 4. The plot shows the convergence of two for the two rules of shuffling $\mathcal{G}_1$ and $\mathcal{G}_2$ for initial probability distributionm $\pi(0) = (1, 0, 0, 0, 0)$.

By figure 4, we can see that $\mathcal{G}_1$ *converges faster than* $\mathcal{G}_2$. *For* $\mathcal{G}_1$ *it takes approximately* 6 *time steps to obtain the uniform equilibrium distribution, and* 19 *time steps for* $\mathcal{G}_2$. *We can derive from this that by using the shuffle rule* $\mathcal{G}_1$ *gets faster mixing than shuffle rule* $\mathcal{G}_2$ *if we initialize the ball at the*

position 0 *for* $\pi(0) = (1, 0, 0, 0, 0)$. *The transition probability matrices we find by solving the FMMC problem can be thought of as the optimal way of moving the ball to get fast mixing.*

CHAPTER 6

Further research

In this chapter we will discuss further reasearch for the FMMC problem.

In this thesis, we have focused on small graphs for the FMMC problem. We have worked with an implemention of interior-point methods which we are able to solve the problem on small graphs. In chapter 4, we solved the FMMC on some small graphs. A type of graph that we tested was cycles. The transition probability matrix that gives fast mixing for such graph, showed to have a simple form. But we were not able to prove the solution analytically as we know is proven for paths in [3]. Although, we where able to solve it numerically, it remains for us to show it analytically. When we compared the time usage of the primal-dual interior-point methods with the CVXOPT package found that our implementation of the interior-point methods had large complexity, not just for modeling the problem, but also the solving the problem. But as the graph became larger we experienced that the also the CVXOPT used quite some time to solve the problem. The methods that we compared shows reasonable solving time for the FMMC on small graphs, but becomes slow when the graph becomes large. It would be of interest if there is possible to make an effective implementation of a convex optimization solver for SDP that can handle large graphs.

As we know, it is difficult to find an exact stopping criterion for the subgradient method, although, we have proven that the method will converge towards the optimal solution, we just do not know when. Therefore, it would be an interesting field of study if there is possible to derive good stopping criterion for the method.

We have also looked at two applications of the FMMC problem, where we looked at two simple examples in shuffling. The graphs of the FMMC problem is considered on undirected graphs, but if it is possible to derive a convex optimization problem on a directed graph that gives fast convergence, we could extend the problem to a lot more wider problem. It may open doors for new applications in areas where directed graphs is much more realistic picture of real life.

APPENDIX A

Graphs and implementation of algorithms

1. NetworkX

NetworkX is open source and distributed with the BCD license. It is a Python language software package for the creation, manipulation, and study of structures, dynamics and functions of complex networks (see [15]).

In this section, we will give an introduction to NetworkX, where we want to explain the basics of representing graphs. So let us get started. The first thing we need to do is to import the package to get access to all the functions, algorithms and datastructures by writing

```
1 import networkx as nx
```

Now that we have imported the package, we are ready to initialize a graph object by

```
1 G1 = nx.Graph() # creates an empty undirected graph
2 G2 = nx.DiGraph() # creates an empty directed graph
3 G3 = nx.MultiGraph() # creates an empty undirected multigraph
4 G4 = nx.MultiDiGraph() # creates an empty directed multigraph
```

We can add to the graph and remove edges from the graph with the following execution

```
1 G = nx.Graph() # initialize a graph object
2
3 G.add_node(0) # creates a node with value 0
4 G.add_node('A') # creates a node with value A
5
6 G.remove_node(0) # removes node with value 0 from the graph
7 G.remove_node('A') # removes node with value A from the graph
```

Weights and labels can be added on the edges

```
1 G.add_edges(0,1) # the default of the weight is 1
2 G.add_edges(0,1,weight=13)
3 G.add_edges('A','B',label='Wall Street')
```

We can also add a bunch of edges in one command by

```
G.add_edges_from([(0,1),(1,2),(2,3)]) # or
G.add_edges_from(zip(range(0,3),range(1,4)))
```

```
G.remove_edge(0,1)
```

To clear the graph we simply execute

```
G.clear()
```

1.1. Drawing graphs. It is fairly easy to draw the graphs generated in NetworkX. We may use different tools to perform this task by either using Graphviz or Matplotlib. Here, we will focus on drawing graphs with Matplotlib, so first we need to import the Matplotlib package by

```
import matplotlib.pyplot as plt
```

Now that we have a drawer ready we will make a small example of how we can draw and visualize the graph.

```
G = nx.Graph()
G.add_path([0,1,2,3])

nx.draw(G)
plt.draw() # draws the graph onto the figure
plt.show() # shows the Matplotlib figure plot
```

2. Implementation of the projected subgradient method

The following Python code is a solver using the simple projected subgradient method described in [2]. The program is used to solve semidefinite programs.

LISTING A.1. Projected subgradient method

```
import numpy as np
import numpy.linalg as la
import networkx as nx

import chains

from copy import copy

def slem(P):
        """
        Finds the second largest eigenvalue modolus (SLEM)
        Attribute:
        P - n x n transition probability matrix

```

```
15          Returns:
16          mu - the second largest eigenvalue modolus
17
18          """
19          eig_vals, eig_vecs = la.eig(P)
20          eig_vals = list(eig_vals)
21          eig_vals.sort()
22          return max(-eig_vals[0], eig_vals[-2]).real
23
24  def f(graph, p):
25          """
26          Objective function of the problem
27          Attribute:
28          p - transition probability vector
29
30          Returns:
31          mu - SLEM value
32
33          """
34          mu = slem(tp_matrix(graph,p))
35          return mu
36
37  def tp_matrix(graph, p):
38          """
39          Finds the transition probability matrix P
40          Attributes:
41          graph - undirected, connected graph in Networkx
42          p - vector of transition probabilities on non-self-loop
                edges
43
44          Returns:
45          P - transition probability matrix
46
47          """
48          edges = graph.edges()
49          n = graph.number_of_nodes()
50          P = np.identity(n)
51          for l in xrange(len(edges)):
52                  E = np.zeros([n,n], dtype=float)
53                  i, j = edges[l]
54                  E[i,j] = 1
55                  E[j,i] = 1
56                  E[i,i] = -1
57                  E[j,j] = -1
58                  P += p[l]*E
59          return P
60
61  def sub(graph, p):
62          """
63          Finds the subgradient step
64          Attributes:
65          graph - undirected, connected graph in NetworkX
```

```
66          p - transition probability vector
67
68          Returns:
69          g - subgradient of P
70
71          """
72          edges = graph.edges()
73          nodes = graph.nodes()
74          m = graph.number_of_edges()
75          n = graph.number_of_nodes()
76          P = tp_matrix(graph,p)
77          g = np.zeros(m)
78
79          eig_vals, eig_vecs = la.eig(P)
80          eig_list = zip(eig_vals, np.transpose(eig_vecs))
81          eig_list.sort(key=lambda x: x[0])
82
83          lambda_2, lambda_n = eig_list[-2][0], eig_list[0][0]
84          if lambda_2 >= -lambda_n:
85                  u = [u_i.real for u_i in eig_list[-2][1]]
86                  for l in xrange(m):
87                          i,j = edges[l]
88                          g[l] = -(u[i] - u[j])**2
89          else:
90                  v = [v_i.real for v_i in eig_list[0][1]]
91                  for l in xrange(m):
92                          i,j = edges[l]
93                          g[l] = (v[i] - v[j])**2
94          return g
95
96
97  def solve(G, p0, max_iter=100,
98                      alpha=lambda k: 1./(np.sqrt(k))):
99          """
100         Minimizes the convex function using the subgradient
                method
101         Attributes:
102         graph - undirected, connected graph in NetworkX
103         p - transition probability vector
104
105         Returns:
106         sol - dictionary of the solution
107         sol['f'] - the best function value of the iterates
108         sol['p'] - the best transition probability vector of
                the iterates
109         sol['fk'] - array of the function iterates
110         sol['iter'] - the iteration number
111         """
112         global p, graph
113         p = p0
114         graph = G
115         edges = graph.edges()
```

```
116         nodes = graph.nodes()
117         n = graph.number_of_nodes()
118         m = graph.number_of_edges()
119         k = 1
120         sol = {'f': f(graph,p),
121                'p': copy(p),
122                'iter' : 0,
123                'fk': np.zeros(max_iter+1)}
124         sol['fk'][0] = f(graph,p)
125         while k <= max_iter:
126                 # subgradient step
127                 g = sub(graph,p)
128                 # sequential projection step
129                 p -= alpha(k)/la.norm(g)*g
130                 for l in range(m): p[l] = max(p[l], 0)
131                 for i in range(n):
132                         I = [l for l in xrange(m) if i in edges
                                [l]]
133                         while sum([p[l] for l in I]) > 1:
134                                 I = [l for l in I if p[l] > 0]
135                                 p_min = min([p[l] for l in I])
136                                 p_sum = sum([p[l] for l in I])
137                                 delta = min(p_min, (p_sum - 1.)
                                    /len(I))
138                                 for l in I: p[l] -= delta
139                 sol['fk'][k] = f(graph,p)
140                 if f(graph,p) < sol['f']:
141                         sol['f'] = f(graph,p)
142                         sol['p'] = copy(p)
143                         sol['iter'] = k
144                 k += 1
145         return sol
146
147 def transition_vector(graph, P):
148         """
149         Finds the transition probability vector of P
150         Attributes:
151         graph - undirected, connected graph in NetworkX
152
153         Returns:
154         p - transition probability vector of P
155
156         """
157         edges = graph.edges()
158         m = graph.number_of_edges()
159         p = np.zeros(m)
160         for l in range(m):
161                 i, j = edges[l]
162                 p[l] = P[i,j]
163         return p
164
165
```

```
166 def optimize(graph,chain=chains.max_deg_matrix,
167                     max_iter=200, alpha=lambda k: 1./np.sqrt(k))
                        :
168         """
169         Solves the FMMC problem on the graph
170         Attribute:
171         graph - undirected, connected graph in NetworkX
172 
173         Optional:
174         chain - inititial probability transition matrix
175         max_iter - maximum iteration number
176         alpha - step size
177 
178         Returns:
179         sol - dictionary of the solution
180         """
181         P = chain(graph)
182         p = transition_vector(graph,P)
183         sol = solve(graph,p,max_iter,alpha)
184         return sol
185 
186 def const_steplength(h):
187         """
188         Constant step length rule
189         Attributes:
190         k - iteration number
191         h - constant
192 
193         Returns:
194         alpha - Python function which takes one attribute
195         """
196         def alpha(k):
197                 return float(h)/la.norm(sub(graph,p))
198         return alpha
199 
200 
201 if __name__ == '__main__':
202         n = 6
203         G = nx.cycle_graph(n)
204         print G.edges()
205         sol = optimize(G,max_iter=5000)
206         mu = sol['f']
207         p = sol['p']
208         i = sol['iter']
209         print mu
210         print tp_matrix(G,p)
211         print i
212 
213         def F(n):
214                 X = np.zeros((n,n))
215 
216                 for i in range(n):
```

```
                        if i-1 < 0:
                                X[n-1,i] = float(n-1)/(2*n)
                        else:
                                X[i-1,i] = float(n-1)/(2*n)
                        if i+1 > n-1:
                                X[0,i] = float(n-1)/(2*n)
                        else:
                                X[i+1,i] = float(n-1)/(2*n)
                        X[i,i] = 1./n
                return X
        print F(n)
        print slem(F(n))
```

LISTING A.2. Maximum-degree chain and Metropolis-Hastings chain

```
import networkx as nx
import numpy as np

def max_deg_matrix(graph):
        """
        Maximum degree chain of a graph
        Attribute:
        graph - undirected, connected graph in NetworkX

        Returns:
        P - transition probability matrix

        """
        n = graph.number_of_nodes()
        P = np.zeros((n,n))
        edges = graph.edges()
        d = [len(graph.neighbors(node)) for node in graph.nodes
            ()]
        d_max = max(d)
        for i,j in edges:
                P[i,j] = 1./d_max
                P[j,i] = P[i,j]
        for i in range(n):
                P[i,i] = 1 - d[i]/float(d_max)
        return P

def tp_rw_matrix(graph):
        """
        Transition probability matrix for random walk.
        Attribute:
        graph - undirected, connected graph in NetworkX

        Returns;
        P - transtion probability matrix

        """
        edges = graph.edges()
```

```
37          n = graph.number_of_nodes()
38          P = np.zeros((n,n))
39          for i,j in edges:
40                  d_i = len(graph.neighbors(i))
41                  d_j = len(graph.neighbors(j))
42                  P[i,j] = 1./d_i
43                  P[j,i] = 1./d_j
44          return P
45
46  def metro_h_matrix(graph,pi_vec=None):
47          """
48          Metropolis-Hastings chain
49          Attribute:
50          graph - undirected, connected graph in NetworkX
51
52          Optional:
53          pi_vec - probability distribution vector
54
55          Returns:
56          P - transition probability matrix
57
58          """
59          n = graph.number_of_nodes()
60          if pi_vec is None:
61                  pi_vec = 1./n*np.ones(n)
62          def mh(graph):
63                  edges = graph.edges()
64                  R = np.zeros((n,n))
65                  P = np.zeros((n,n))
66                  P_rw = tp_rw_matrix(graph)
67                  for i in range(n):
68                          for j in range(n):
69                                  R[i,j] = (pi_vec[j]*P_rw[j,i])/
                                        \
70                                           (pi_vec[i]*P_rw[i,j])
71                  for i,j in edges:
72                          P[i,j] = P_rw[i,j]*min(1,R[i,j])
73                          P[j,i] = P[i,j]
74                  for i in range(n):
75                          s = 0
76                          for k in graph.neighbors(i):
77                                  s += P_rw[i,k]*(1 - min(1,R[i,k
                                        ]))
78                          P[i,i] = P_rw[i,i] + s
79                  return P
80          return mh
81
82  def metro_h_matrix_uniform(graph):
83          """
84          Metropolis-Hastings chain for uniform distribution.
85          Attribute:
86          graph - undirected, connected graph in NetworkX
```

```
87
88          Optional:
89          pi_vec - probability distribution vector
90
91          Returns:
92          P - transition probability matrix
93
94          """
95          edges = graph.edges()
96          n = graph.number_of_nodes()
97          P = np.zeros((n,n))
98          for i,j in edges:
99                  d_i = len(graph.neighbors(i))
100                 d_j = len(graph.neighbors(j))
101                 P[i,j] = min(1./d_i, 1./d_j)
102                 P[j,i] = P[i,j]
103         for i in range(n):
104                 s = 0
105                 d_i = len(graph.neighbors(i))
106                 for k in graph.neighbors(i):
107                         d_k = len(graph.neighbors(k))
108                         s += max(0,1./d_i-1./d_k)
109                 P[i,i] = s
110         return P
111
112 if __name__ == '__main__':
113         n = 4
114         G = nx.path_graph(n-1)
115         f = metro_h_matrix(G)
116         print f
117         f = max_deg_matrix(G)
118         print f
```

3. Implementation of the interior-point methods for SDP

LISTING A.3. Modelling the FMMC problem for the primal-dual interior-point method

```
1  from pd_interior_sdp import XZ, XZZX
2
3  import networkx as nx
4  import numpy as np
5  import time
6
7  def diag2(*elements):
8          """
9          creates the block diagonals of the elements given
10         Attribute:
11         elements - scalars or matrices to set on the block
               diagonal
12
13         Returns:
```

```
        X - a square matrix where the elements are placed at
            the
            block diagonal.
        """
        s = 0
        for i in range(len(elements)):
                if type(elements[i]) == np.ndarray:
                        s += len(elements[i])
                else:
                        s += 1

        X = np.zeros((s,s))
        m = 0
        for i in range(len(elements)):
                x = elements[i]
                k = 0
                if type(elements[i]) == np.ndarray:
                        k = len(x)
                else:
                        k = 1
                X[m:m+k,m:m+k] = x
                m += k
        return X

def fmmc(graph):
        """
        create the equality contraints given the graph
        Attribute:
        graph - undirected, connected graph in NetworkX

        Returns:
        A - list of matrices corresponding to the equality
            constraints of
            the FMMC problem
        b - list of scalars corresponding to the equlity
            contraints of
            the FMMC problem
        the problem.
        """
        n = graph.number_of_nodes()
        A = []
        b = []
        Z = np.zeros((n,n))
        non_edges = [e for e in nx.non_edges(graph)]

        # symmetry
        for i in range(1,n):
                for j in range(i):
                        E = np.zeros(n*n)
                        E[i*n+j] = 1
                        E[j*n+i] = -1
                        E = E.reshape(n*n)
```

```
63                          A.append(diag2(Z,Z,np.diag(E),0))
64                          b.append(0)
65
66          # row/column sum
67          for i in range(n):
68                  E = np.zeros((n,n))
69                  E[i] = np.ones(n)
70                  E = E.reshape(n*n)
71                  A.append(diag2(Z,Z,np.diag(E),0))
72                  b.append(1)
73
74          # edges not in the graph is set to zero
75          for i,j in non_edges:
76                  E = np.zeros(n*n)
77                  E[j*n+i] = 1
78                  A.append(diag2(Z,Z,np.diag(E),0))
79                  b.append(0)
80
81          # M1 = sI - P + 1./n*11
82          for i in range(n):
83                  for j in range(n):
84                          E = np.zeros((n,n))
85                          E[i,j] = 1
86                          if i == j:
87                                  A.append(diag2(E,Z,np.diag(E.
                                        reshape(n*n)),-1))
88                          else:
89                                  A.append(diag2(E,Z,np.diag(E.
                                        reshape(n*n)),0))
90                          b.append(1./n)
91
92          # M2 = sI + P - 1./n*11
93          for i in range(n):
94                  for j in range(n):
95                          E = np.zeros((n,n))
96                          E[i,j] = 1
97                          if i == j:
98                                  A.append(diag2(Z,E,np.diag(-E.
                                        reshape(n*n)),-1))
99                          else:
100                                 A.append(diag2(Z,E,np.diag(-E.
                                        reshape(n*n)),0))
101                         b.append(-1./n)
102         return A, b
103
104 def optimize(graph, method='XZ', tol=1E-7, MAX_ITER=100, sigma
        =0.25, tau=0.8):
105         """
106         optimizes the FMMC problem given a graph
107         Attribute:
108         graph - undirected, connected graph
109
```

```
110         Optional:
111         method - either XZ- or XZ+ZX-method can be selected as
                solver
112         tol - tolerance of the duality gap
113         MAX_ITER - maximum number of iteration of the method
114         sigma - parameter
115         tau - paramter
116
117         Returns:
118         sol - dictionary of the solution which contains:
119         sol['time'] - time usage of the method
120         sol['X'] - primal solution matrix
121         sol['y'] - dual solution vector
122         sol['Z'] - dual solution matrix
123         """
124         n = graph.number_of_nodes()
125         Ak, bk = fmmc(graph)
126         m = len(bk)
127         X = np.eye((n+1)**2)
128         y = np.zeros(m)
129         Z = np.eye((n+1)**2)
130         C = np.zeros(((n+1)**2,(n+1)**2))
131         C[-1,-1] = 1
132         if method == 'XZZX':
133                 sol = XZZX(Ak, bk, C, X, y, Z, tol, MAX_ITER,
                        sigma, tau)
134         else:
135                 sol = XZ(Ak, bk, C, X, y, Z, tol, MAX_ITER,
                        sigma, tau)
136         return sol
137
138 def get_P(X, n):
139         """
140         extract the transition probabability matrix from the
                primal matrix
141         Attribute:
142         X - primal solution
143         n - number of nodes in the graph
144
145         Returns:
146         P - transition probabability matrix of the FMMC
147         """
148         P = np.zeros((n,n))
149         k = 2*n
150         for i in range(n):
151                 for j in range(n):
152                         P[i,j] = X[k,k]
153                         k += 1
154         return P
155
156 def slem(P):
157         """
```

```
158         calculates the second largest eigenvalue modulus
159         attribute:
160         P - transition matrix of the graph
161
162         """
163         n = len(P)
164         v, w = np.linalg.eig(P)
165         v.sort()
166         return max(abs(v[0]),abs(v[-2]))
167
168 if __name__ == '__main__':
169         import networkx as nx
170         n = 5
171         G = nx.star_graph(n-1)
172         G.add_star(range(n))
173         sol = optimize(G,method='XZZX', tol=1E-3)
174         print sol['time']
175         sol = optimize(G,method='XZ', tol=1E-3)
176         print sol['time']
177         X_opt = sol['X']
178         P = get_P(X_opt,n)
179         s = sol['X'][-1,-1]
180         print s
181         print slem(P)
```

LISTING A.4. Primal-dual interior-point methods

```
1  import numpy as np
2
3  def nvec(mat):
4          """
5          Reshape a matrix into a vector such that
6          the columns of the matrix are stacked
7          attribute:
8          mat - matrix
9
10         returns:
11         vec - vectorization of the matrix mat
12
13         """
14         n = mat.size
15         vec = np.transpose(mat).reshape(n)
16         return vec
17
18 def svec(mat):
19         """
20         Transform a symmetric matrix into a vector
21         attribute:
22         mat - symmetric matrix
23
24         returns:
25         vec - symmetric vectorization of the matrix
26
```

```
        """
        n = len(mat)
        vec = np.zeros(n*(n+1)/2)
        k = 0;
        for i in range(n):
                for j in range(i+1):
                        if i == j:
                                vec[k] = mat[i,j]
                        elif i < j:
                                vec[k] = np.sqrt(2)*mat[i,j]
        return vec

def kron_sym(A, B):
        """
        Symmetric kronecker product
        Attributes:
        A - symmetric matrix
        B - symmetric matrix

        Returns:
        M - kronecker product of A and B

        """
        M = 0.5*(np.kron(A,B) + np.kron(B,A))
        return M

def mat(vec):
        """
        Transform a matrix into a vector
        Attribute:
        vec - vector with n**2 elements

        Returns:
        M - matrix

        """
        n = len(vec)
        m = int(np.sqrt(n))
        matrix = vec.reshape((m,m))
        matrix = np.transpose(matrix)
        return matrix

def steplength(X, dX, tau):
        """
        Calculates the steplength of the interior point method
        Attributes:
        X - matrix
        dX - matrix
        tau - parameter value

        Returns:
        alpha - the step length
```

```
79  |                                                                    |
80  |         """                                                        |
81  |         L = np.linalg.cholesky(X)                                  |
82  |         L_inv = np.linalg.inv(L)                                   |
83  |         w, v = np.linalg.eig(np.dot(-np.dot(L_inv, dX),            |
84  |                              np.transpose(L_inv)))                 |
85  |         w.sort()                                                   |
86  |         lambda_max = w[-1]                                         |
87  |         alpha_hat = 1./lambda_max                                  |
88  |         alpha = min(1, tau*alpha_hat)                              |
89  |         return alpha.real                                          |
90  |                                                                    |
91  | def search_XZ(n, m, X, y, Z, mu):                                  |
92  |         """                                                        |
93  |         Finds the search direction of the XZ-method.               |
94  |         Attributes:                                                |
95  |         n - dimension of X                                         |
96  |         m - number of constraints                                  |
97  |         X - primal variable matrix                                 |
98  |         y - dual variable vector                                   |
99  |         Z - dual variable matrix                                   |
100 |         mu - scalar                                                |
101 |                                                                    |
102 |         Returns:                                                   |
103 |         dX - primal step                                           |
104 |         y - dual step                                              |
105 |         dZ - dual step                                             |
106 |                                                                    |
107 |         """                                                        |
108 |         x = X.reshape(n*n)                                         |
109 |         I = np.eye(n)                                              |
110 |         rp = b - np.dot(A,x)                                       |
111 |         Rd = C - Z - mat(np.dot(np.transpose(A),y))                |
112 |         Rc = mu*I - np.dot(X,Z)                                    |
113 |         rd = nvec(Rd)                                              |
114 |         rc = nvec(Rc)                                              |
115 |                                                                    |
116 |         E = np.kron(Z,I)                                           |
117 |         E_inv = np.linalg.inv(E)                                   |
118 |         A_t = np.transpose(A)                                      |
119 |         F = np.kron(I,X)                                           |
120 |         M = np.dot(np.dot(A,E_inv),                                |
121 |                    np.dot(F,A_t))                                  |
122 |         dy = np.linalg.solve(M,rp + np.dot(np.dot(A,E_inv),        |
123 |                                            np.dot(F,rd) - rc))     |
124 |         dx = -np.dot(E_inv,np.dot(F,rd - np.dot(A_t,dy)) - rc)     |
125 |         dz = rd - np.dot(np.transpose(A), dy)                      |
126 |         dX = mat(dx)                                               |
127 |         dZ = mat(dz)                                               |
128 |         return dX, dy, dZ                                          |
129 |                                                                    |
130 | def search_XZZX(n, m, X, y, Z, mu):                                |
```

```
131         """
132         Finds the search direction of the XZ+ZX-method.
133         Attributes:
134         n - dimension of X
135         m - number of constraints
136         X - primal variable matrix
137         y - dual variable vector
138         Z - dual variable matrix
139         mu - scalar
140 
141         Returns:
142         dX - primal step
143         y - dual step
144         dZ - dual step
145 
146         """
147         x = X.reshape(n*n)
148         I = np.eye(n)
149         rp = b - np.dot(A, x)
150         Rd = C - Z - mat(np.dot(np.transpose(A), y))
151         Rc = mu*I - 0.5*(np.dot(X,Z) + np.dot(Z,X))
152         rd = nvec(Rd)
153         rc = nvec(Rc)
154 
155         E = kron_sym(Z,I)
156         F = kron_sym(X,I)
157         E_inv = np.linalg.inv(E)
158         A_t = np.transpose(A)
159         M = np.dot(np.dot(A,E_inv),
160                    np.dot(F,A_t))
161         dy = np.linalg.solve(M,rp + np.dot(np.dot(A,E_inv),
162                                            np.dot(F,rd) - rc))
163         dx = -np.dot(E_inv,np.dot(F,rd - np.dot(A_t,dy)) - rc)
164         dz = rd - np.dot(A_t,dy)
165         dX = mat(dx)
166         dZ = mat(dz)
167         return dX, dy, dZ
168 
169 def XZ(Ak, bk, C0, X0, y0, Z0, tol=1E-2, MAX_ITER=100, sigma
        =0.25, tau=0.5):
170         """
171         Finds the search direction of the XZ-method.
172         Attributes:
173         Ak - list of matrices of the problem
174         bk - list of scalars defining the problem
175         C0 - matrix of the objective function
176         X0 - symmetric, positive semidefinite matrix in R^n
177         y0 - vector in R^m
178         Z0 - symmetric, positive semidefinite matrix in R^n
179 
180         Optional:
181         tol - tolerance of the duality gap
```

```
182         MAX_ITER - maximum iterations
183         sigma - parameter value
184         tau - parameter value
185
186         Returns:
187         sol - dictionary of the solution
188         sol['X'] - primal optimal matrix
189         sol['y'] - dual optimal vector
190         sol['Z'] - dual optimal matrix
191         sol['iter'] - number of iterations
192
193         """
194         m = len(bk)
195         n = len(C0)
196         global A, b, C
197         C = C0
198         A = np.zeros((m,n*n))
199         for k in range(m):
200                 A[k] = Ak[k].reshape(n*n)
201         b = np.array(bk)
202
203         k = 0
204         sol = {'X':X0,'y':y0,'Z':Z0,'iter':k}
205         X, y, Z = X0, y0, Z0
206         while k < MAX_ITER:
207                 if 0 <= np.sum(Z*X) <= tol:
208                         print 'optimal solution found!'
209                         print 'iterations=%d' % k
210                         sol['X'] = X
211                         sol['y'] = y
212                         sol['Z'] = Z
213                         sol['iter'] = k
214                         return sol
215                 mu = sigma*np.sum(X*Z)/n
216                 dX, dy, dZ = search_XZ(n, m, X, y, Z, mu)
217                 dX = 0.5*(dX + np.transpose(dX))
218                 alpha = steplength(X, dX, tau)
219                 beta = steplength(Z, dZ, tau)
220                 X += alpha*dX
221                 y += beta*dy
222                 Z += beta*dZ
223                 k += 1
224         print 'solution not found'
225         return sol
226
227 def XZZX(Ak, bk, C0, X0, y0, Z0, tol=1E-7, MAX_ITER=100, sigma
        =0.25, tau=0.5):
228         """
229         Finds the search direction of the XZ+ZX-method.
230         Attributes:
231         Ak - list of matrices of the problem
232         bk - list of scalars defining the problem
```

```
        C - matrix of the objective function
        X0 - symmetric, positive semidefinite matrix in R^n
        y0 - vector in R^m
        Z0 - symmetric, positive semidefinite matrix in R^n

        Optional:
        tol - tolerance of the duality gap
        MAX_ITER - maximum iterations
        sigma - parameter value
        tau - parameter value

        Returns:
        X - primal optimal matrix
        y - dual optimal vector
        Z - dual optimal matrix

        """
        m = len(bk)
        n = len(C0)
        global A, b, C
        C = C0
        A = np.zeros((m,n*n))
        for k in range(m):
                A[k] = Ak[k].reshape(n*n)
        b = np.array(bk)
        k = 0
        sol = {'X':X0,'y':y0,'Z':Z0,'iter':k}
        X, y, Z = X0, y0, Z0
        while k < MAX_ITER:
                if 0 <= np.sum(Z*X) <= tol:
                        print 'optimal solution found!'
                        print 'iterations=%d' % k
                        sol['X'] = X
                        sol['y'] = y
                        sol['Z'] = Z
                        sol['iter'] = k
                        return sol
                mu = sigma*np.sum(X*Z)/n
                dX, dy, dZ = search_XZZX(n, m, X, y, Z, mu)
                alpha = steplength(X, dX, tau)
                beta = steplength(Z, dZ, tau)
                X += alpha*dX
                y += beta*dy
                Z += beta*dZ
                k += 1
        print 'solution not found'
        return sol

if __name__ == '__main__':
        # test run on a small semidefinite program example
        n = 2
        m = 1
```

```
285         A1 = np.eye(n)
286         b1 = 1
287
288         C = np.zeros((n,n))
289         C[0,0] = 2
290         C[1,0] = 1
291         C[0,1] = 1
292         X = 0.5*np.eye(n)
293
294         y = -1
295
296         Z = np.zeros((n,n))
297         Z[0,0] = 1
298         Z[1,0] = 0
299         Z[0,1] = 0
300         Z[1,1] = 1
301
302         sol = XZZX([A1], [b1], C, X, y, Z,tol=1E-7)
303         X_opt, y_opt, Z_opt = sol['X'], sol['y'], sol['Z']
304         X_exact = np.zeros((n,n))
305         X_exact[0,0] = (2 - np.sqrt(2))/4
306         X_exact[0,1] = -1./(2*np.sqrt(2))
307         X_exact[1,0] = -1./(2*np.sqrt(2))
308         X_exact[1,1] = (2 + np.sqrt(2))/4
309
310         print 'X_opt=\n', X_opt
311         print 'X_exact=\n', X_exact
312         print 'obj=\n', np.sum(C*X_opt)
313         print 'obj_exact=\n', np.sum(C*X_exact)
```

4. Implementation of generating random probability distribution

LISTING A.5. Python code for generating random probability distribution

```
1  import numpy as np
2
3  def rand_dist(n):
4          """
5          Generate a random probability distribution
6          Attribute:
7          n - number of states of the Markov chain
8
9          Returns:
10         u - probability distribution
11         """
12         u = np.zeros(n)
13         for i in range(n-1):
14                 u[i] = np.random.uniform(high=1-sum(u))
15         u[n-1] = 1-sum(u)
16         return u
17
18 if __name__ == '__main__':
```

```
19          n = 5
20          u = rand_dist(n)
21          print u
```

5. Implementation of generating random graphs

LISTING A.6. Python code for generating random graphs

```
1  import networkx as nx
2  import numpy as np
3
4  def midpoint(a, b):
5          return (a+b)/2.
6
7  def smallest_c(R, cmin, cmax, tol=1E-7):
8          """
9          Finds the smallest c value that gives a connected graph
10         with a binary search
11         Attributes:
12         R - symmetric matrix
13         cmin - minimum c value
14         cmax - maximum c value
15
16         Optional:
17         tol - tolerance
18
19         Returns:
20         cmax - smallest c value
21         """
22         while (cmax-cmin > tol):
23                 cmid = midpoint(cmin,cmax)
24                 G = nx.Graph(data=(R<=cmid))
25                 if nx.is_connected(G):
26                         cmax = cmid
27                 else:
28                         cmin = cmid
29         return cmax
30
31 def generate_R(n):
32         """
33         Generates a symmetric matrix with n vertices.
34         Attribute:
35         n - number of vertices
36
37         Returns:
38         R - symmetric matrix
39         """
40         R = np.random.uniform(size=(n,n))
41         for i in range(n):
42                 for j in range(i):
43                         R[j,i] = R[i,j]
44         return R
45
```

```
46 if __name__ == '__main__':
47         R = generate_R(50)
48         cmin = smallest_c(R,0,1)
```

6. CVXOPT

CVXOPT is a free software package for convex optimization based on the Python programming language. The package makes it easier to develope software for convex optimization applications by building on the standard library of Python and on the strengths as high-level programming language.

Here we will give a short introduction to the function *cvxopt.solvers.sdp* that will help us solve semidefinite programming. The function solve primal and dual semidefinite programs on the form

$$\begin{aligned} &\text{minimize} && c^T x \\ &\text{subject to} && G_0 x + s_0 = h_0 \\ &&& G_k x + \text{vec}(s_k) = \text{vec}(h_k),\ k = 1, \ldots, N \\ &&& Ax = b \\ &&& s_k \succeq 0,\ k = 0, \ldots, N \end{aligned} \tag{74}$$

and

$$\begin{aligned} &\text{maximize} && -h_0^T z_0 - \sum_{k=1}^{N} \text{tr}(h_k z_k) - b^T y \\ &\text{subject to} && G_0^T z_0 + \sum_{k=1}^{N} G_k^T \text{vec}(z_k) + A^T y + c = 0 \\ &&& z_k \succeq 0,\ k = 0, \ldots, N \end{aligned}$$

We note here that the primal variables pair s_0, h_0 are vectors and s_k, h_k for $k = 1, \ldots, N$ are matrices. And similar for the dual variables in the dual program, z_0 is a vector and z_k for $k = 1, \ldots, N$ are matrices. $s_0 \succeq 0$ means that the elements of s_0 are non-negative and $s_k \succeq 0$ for $k = 1, \ldots, N$ are positive semidefinite matrices. $\text{vec}(A)$ means that we vectorize the matrix A by stacking the columns of the matrix and $\text{tr}(AB)$ is the trace of the matrix product AB which is defined as $\text{tr}(AB) = \sum_i \sum_j A_{ij} B_{ij}$. The parameters we have to assign to model the problem we want to solve are c,A,b, and G_k and h_k for $k = 0, 1, \ldots, N$.

We will now provide an example of the usage of the function by looking at a simple problem. We will write the code in Python, but first we define the problem.

Listing A.7. Modelling FMMC for solving with CVXOPT

```
1 from cvxopt import solvers, matrix
2 import networkx as nx
3 import numpy as np
4
5 def fmmc(graph):
6         """
```

```
        Solves the FMMC problem formulated as a sdp using
            cvxopt.
        Attribute:
        graph - undirected NetworkX graph

        Returns:
        P - optimal transition probability matrix
        s - optimal SLEM value mu
        """
        n = graph.number_of_nodes()
        non_edges = [e for e in nx.non_edges(graph)]

        m = n+0.5*n*(n-1)+len(non_edges)
        A = np.zeros((m,n**2+1))
        b = np.zeros(m)
        c = np.zeros(n**2+1)
        G1 = np.zeros((n**2,n**2+1))
        G2 = np.zeros((n**2,n**2+1))
        h1 = (1./n)*np.ones((n,n))
        h2 = -(1./n)*np.ones((n,n))

        k = 0
        # row/sum of P, P1=1
        for i in range(n):
                A[k,i*n:(i+1)*n] = np.ones(n)
                k += 1
        # symmetry of P, P=P^T
        for i in range(1,n):
                for j in range(i):
                        A[k,i+j*n] = 1
                        A[k,j+i*n] = -1
                        k += 1
        # P_ij = 0 for (i,j) not in graph
        for i,j in non_edges:
                A[k,i*n+j] = 1
                k += 1

        b[0:n] = np.ones(n)

        # P - sI <= (1/n)11^T
        # -(P + sI) <= -(1/n)11^T
        for i in range(n):
                for j in range(n):
                        if i == j:
                                G1[i*n+j,-1] = -1
                                G2[i*n+j,-1] = -1
                        G1[i*n+j,i*n+j] = 1
                        G2[i*n+j,i*n+j] = -1
        G = [matrix(G1), matrix(G2)]
        h = [matrix(h1), matrix(h2)]

        # objective function c^Tx
```

```
58          c[-1] = 1
59          c = matrix(c)
60          I = matrix(np.eye(n**2+1))
61          h0 = matrix(np.zeros(n**2+1))
62          sol = solvers.sdp(c, Gl=matrix(-I),
63                               hl=h0, Gs=G,
64                               hs=h, A=matrix(A),
65                               b=matrix(b))
66
67          P = np.reshape(sol['x'][:-1],(n,n))
68          s = sol['x'][-1]
69          return P, s
70
71  if __name__ == '__main__':
72          n = 3
73          G = nx.Graph()
74          G.add_star(range(n))
75          P,s = fmmc(G)
76          print P, s
```

APPENDIX B

Monte Carlo simulation

1. Simulation: Random walk on a graph

In this section we will simulate what we mean by a random walk on a graph.

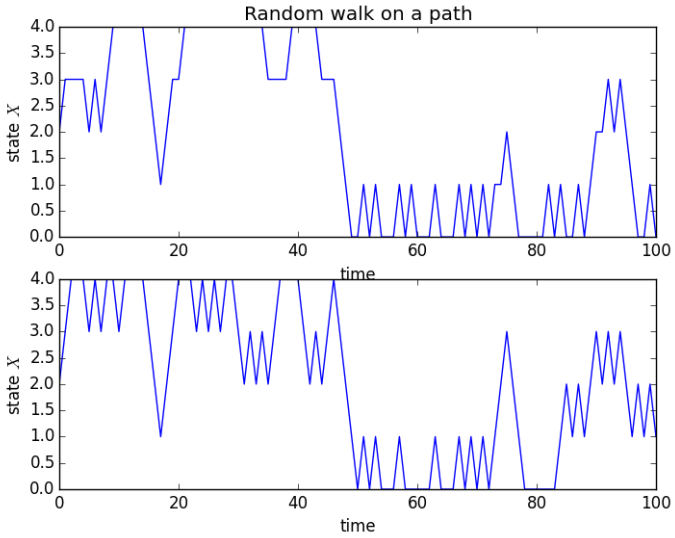

FIGURE 1. The figure shows random walk on a path with two different transition probability matrices.

In figure 1 we have simulated two random walks on a path. We have used two different transition probability matrices and drawn the same random variables in both cases, to compare the progress of a walk based on the matrix. We can see that walk transit more for the bottom plot than the top plot.

We can use Monte Carlo method to calculate the probability distribution for random walk on a graph. To find the probability of being in a state, we simulate many random walks, count the number of occurences, and divide by the number of walks.

LISTING B.1. Simulation of random walk on a graph

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import random
5
```

```
6  def compute_probability(graph, time, num_exp):
7          """
8          Monte Carlo simulation of random walk on a graph.
9          Computes the probability distribution for the random
10         walk.
11         Attributes:
12         graph - undirected, connected graph
13         time - time steps
14         num_exp - number of experiments
15
16         Returns:
17         P - probability distribution for each time step
18         """
19         N = graph.number_of_nodes()
20         P = np.zeros((time+1,N))
21         n = 0
22         while n < num_exp:
23                 states = random_walk(graph, time)
24                 for t in range(time+1):
25                         P[t,states[t]] += 1
26                 n += 1
27         P = 1./num_exp*P
28         return P
29
30
31 def random_walk(graph,time, X0=0):
32         """
33         Computes a random walk on a graph
34         Attributes:
35         graph - undirected, connected graph in NetworkX
36         time - time steps
37
38         Returns:
39         states - array of states for each time steps
40         """
41         n = graph.number_of_nodes()
42         X = X0
43         states = np.zeros(time+1)
44         states[0] = X0
45         P = np.zeros((n,n))
46         for i,j in graph.edges():
47                 P[i,j] = graph[i][j]['weight']
48                 P[j,i] = graph[i][j]['weight']
49
50         for t in range(1,time+1):
51                 X = int(np.random.choice(n, size=1, p=P[X]))
52                 states[t] = X
53         return states
54
55 if __name__ == '__main__':
56         n = 6
57         graph = nx.Graph()
```

```
58          graph.add_path(range(n))
59          graph.add_edge(0,0,weight=0.5)
60          graph.add_edge(n-1,n-1,weight=0.5)
61          for i in range(n-1):
62                  graph.add_edge(i,i+1,weight=0.5)
63
64          T = 100
65          num_exp = 1000
66
67          states = random_walk(graph, T)
68          P = compute_probability(graph, T, num_exp)
69
70          plt.plot(range(T+1), states, '-x')
71          plt.show()
```

Bibliography

[1] Martin S. Andersen, Joachim Dahl, and Lieven Vandenberghe. *CVXOPT*. cvxopt.org.

[2] Stephen Boyd, Persi Diaconis, and Lin Xiao. "Fastest mixing Markov chain on a graph". In: *SIAM Rev.* 46.4 (2004), pp. 667–689. ISSN: 0036-1445. DOI: 10.1137/S0036144503423264. URL: http://dx.doi.org/10.1137/S0036144503423264.

[3] Stephen Boyd et al. "Fastest mixing Markov chain on a path". In: *Amer. Math. Monthly* 113.1 (2006), pp. 70–74. ISSN: 0002-9890. DOI: 10.2307/27641840. URL: http://dx.doi.org/10.2307/27641840.

[4] J.A. Bondy and U.S.R. Murty. *Graph Theory*. Macmilian Press Ltd., 2008.

[5] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. 2 Revised. American Mathematical Society, 1997.

[6] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[7] Stephen Boyd, Lin Xiao, and Almir Mutapcic. *Subgradient Methods*. https://web.stanford.edu/class/ee392o/subgrad_method.pdf. Notes for EE392o, Stanford University. 2003.

[8] David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov chains and mixing times*. With a chapter by James G. Propp and David B. Wilson. American Mathematical Society, Providence, RI, 2009, pp. xviii+371. ISBN: 978-0-8218-4739-8.

[9] Farid Alizadeh, Jean-Pierre A. Haeberly, and Michael L. Overton. "Primal-dual interior-point methods for semidefinite programming: convergence rates, stability and numerical results". In: *SIAM J. Optim.* 8.3 (1998), pp. 746–768. ISSN: 1052-6234. DOI: 10.1137/S1052623496304700. URL: http://dx.doi.org/10.1137/S1052623496304700.

[10] Carl D. Meyer. *Matrix Analysis and Applied Linear Algebra*. 2000.

[11] David C. Lay. *Linear Algebra and Its Applications*. 3rd ed. 2006.

[12] Luca Trevisan. *Graph Patitioning and Expanders*. http://theory.stanford.edu/~trevisan/cs359g/lecture02.pdf.

[13] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization, Solutions Manual*. 2006.

[14] Torgeir Børresen. "The Fastest Mixing Markov Chain Problem". MA thesis. University of Oslo, 2014.

[15] *NetworkX Documentation*. http://networkx.github.io/documentation/networkx-1.9.1/overview.html.

[16] Ben-Tal Nemirovski. *Modern Convex Optimization*.

[17] Lieven Vandenberghe and Stephen Boyd. “Semidefinite programming”. In: *SIAM Rev.* 38.1 (1996), pp. 49–95. ISSN: 0036-1445. DOI: 10.1137/1038003. URL: http://dx.doi.org/10.1137/1038003.